PL/I METAMODEL

Shyh-Mei F. Ho

Eugene Dong

Alan Roeder Elderon

Tony Y. Tsai

Cross Reference to Related Application

This application claims the benefit under Title 35, United States Code, Sections 111(b) and 119(e), relating to Provisional Patent Applications, of the filing date of United States Provisional Patent Application Serial Number 60/223,671 filed August 8, 2000 of Steven A. Brodsky and Shyh-Mei F. Ho for **EAI Common Application Metamodel**.

This application is also related to the following United States Patent Applications, filed on even date herewith:

COMMON APPLICATION METAMODEL by Shyh-Mei Ho, Stephen Brodsky, and James Rhyne,

COBOL METAMODEL by Shyh-Mei Ho, Nick Tindall, James Rhyne, Tony Tsai, Peter Elderon, and Shahaf Abileah.

HIGH LEVEL ASSEMBLER METAMODEL by Shyh-Mei Ho, John Ehrman, Benjamin Sheats, and Jenny Hung.

TYPE DESCRIPTOR METAMODEL by Shyh-Mei Ho, James Rhyne, Peter Elderon, Nick Tindal, and Tony Tsai.

SVL920010036US1

IMS TRANSACTION MESSAGES METAMODEL by Shyh-Mei Ho
and Shahaf Abileah.

IMS-MFS (MESSAGE FORMAT SERVICE) METAMODEL by
5   Shyh-Mei Ho, Benjamin Sheats, Elvis Halcrombe, and Chenhuei J.
Chiang.

CICS-BMS (BASIC MESSAGE SERVICE) METAMODEL by Shyh-
Mei Ho, Andy Krasun, and Benjamin Sheats.

10

Field of the Invention

The invention relates to exchanging instructions and/or data between applications to
signal readiness to transfer, exchange, or process data, or to establish at least one or more
15   parameters for transferring data between the applications, and controlling the parameters
in order to facilitate data transfer and communication. The invention further relates to
integrating dissimilar applications one executing within one platform and another
executing in another platform, e.g., multiple computers, multiple operating systems,
multiple application components, multiple development environments, multiple
20   deployment environments, or multiple testing and processing, establishing a dialog (e.g.,
a negotiation) with one another in order to establish connectivity for transferring data
and/or instructions between the applications so as to facilitate performing tasks on the
data or portions thereof to accomplish an overall goal. The parameters may include one
or more of format, data types, data structures, or commands.

25

Background

2

The growth of e-business has created a significant need to integrate legacy applications and bring them to the Internet This is because the current trend for new applications is to embrace Web standards that simplify end user application construction and scalability. Moreover, as new applications are created, it is crucial to seamlessly integrate them with

5   existing systems while facilitating the introduction of new business processes and paradigms.

Integrating new applications with existing applications is especially critical since industry analysts estimate that more than seventy percent of corporate data, including data highly

10   relevant to e-commerce, lives on mainframe computers. Moreover, while many e-commerce transactions are initiated on Windows, Mac, and Linux end user platforms, using a variety of Web browsers, and go through Windows NT and Unix servers, they are ultimately completed on mainframe computers, running mainframe applications, and impacting data stored in mainframe databases.

15

There are e-business pressures to integrate server level applications and bring them to the Internet. However, there is no complete and easy mechanism to integrate or e-business enable the applications. Integration, whether through messaging, procedure calls, or database queries, is key to solving many of today's business problems.

20

Integrating legacy applications with new software is a difficult and expensive task due, in large part, to the need to customize each connection that ties together two disparate applications. There is no single mechanism to describe how one application may allow itself to be invoked by another.

25

One consequence is an e-commerce environment of multiple applications, developed by multiple development teams, running on different platforms, with different data types, data structures, commands, and command syntax's. This environment is stitched together with application program interfaces and connectors. Connectors are an essential part of

30   the total application framework for e-commerce. Connectors match interface requirements of disparate applications and map between disparate interfaces.

This growing interconnection of old and new software systems and applications, has led to various middle ware applications and connector applications, interface specifications, interface definitions, and code, especially for the interconnection and interaction of

5 markup languages (such as HTML, XML, Dynamic HTML, WML, and the like), through object oriented languages such as SmallTalk and C++, with languages of legacy application server applications (such as PL/I and COBOL). These interface specifications, definitions, and code should apply across languages, tools, applications, operating systems, and networks so that an end user experiences the look, feel, and

10 responses of a single, seamless application at her terminal. Instead, the proliferation of standards, protocols, specifications, definitions, and code, e.g., Common Object Request Broker (CORBA), Common Object Model (COM), Object Linking and Embedding (OLE), SOM, ORB Plus, Object Broker, Orbix, has instead created an e-commerce "Tower of Babel."

15

Examples of application integration are ubiquitous: from installing an ERP system, to updating an Operational Data Store (ODS) with IMS transactions or invoking CRM systems from MQSeries; each of these requires the same basic steps. First, a user must find the entity she wants to communicate with, then she must figure out how to invoke

20 the entity, and finally she must provide translation from one native representation to another. Today, these steps usually require manual investigation and hand coding - and leave the developers with a rat's-nest of hard-to-maintain connections between applications.

25 Attempts to remedy this situation involve application program interfaces and connectors, which are frequently built on Interface Definition Languages. Interface Definition Languages are declarative, defining application program interfaces, and, in some cases, issues such as error handling. Most Interface Definition Languages are a subset of C++, and specify a component's attributes, the parent classes that it inherits from, the

30 exceptions that it raises, the typed events that it emits, the methods its interface supports, input and output parameters, and data types. The goal of Interface Definition Languages

4

within connectors is to enable collaboration between dissimilar applications without hard coded application program interfaces.

Ideally, the interface definition language, and the connector of which it is a part, should
5   facilitate full run-time software application collaboration through such features as

- Method invocation with strong type checking,
- Run-time method invocation with greater flexibility and run time binding,
- High level language binding, with the interface separated from the
10   implementation.
- An interface repository containing real time information of server functions and parameters.

Additionally, the connector and its interface definition language, should be fast, efficient,
15   scalable, portable, support metaclasses, support syntactic level extensions, and support semantic level extensions.

Summary of the Invention

20   The problems associated with integrating new applications, for example, e-commerce applications, with legacy applications are obviated by the Common Application Metamodel tool, method, and system described herein. The Common Application Metamodel method, tool, and system of the invention facilitate tooling solutions, data translation, and communication and collaboration between dissimilar and disparate
25   applications, as well as full run-time software application collaboration through an interface with the application server interface domain. This is accomplished through metadata interchange information, method invocation with strong type checking, run-time method invocation, run time binding, and high level language binding, with the interface separated from the implementation, and an interface repository containing real
30   time information of client and server interface parameters.

SVL920010036US1

Additionally, the tool, method, and system of the invention provide fast, efficient, and scalable interconnectivity independently of any tool or middleware, are reusable and portable, and support metaclasses, syntactic level extensions, and semantic level extensions, and are independent of any particular tool or middleware.

5

The Common Application Metamodel tool, method, and system is especially useful for providing a data transformer that is bi-directional between a client application and a server application, transmitting commands and data both ways between, for example, a Java, HTML, XML, C, or C++ application and a PL/I application.

10

One embodiment of the invention is a method of processing a transaction on or between an end user application and one or more application servers. The method comprises the steps of initiating the transaction on the end user application in a first language with a first application program, transmitting the transaction to the server, and converting the

15 transaction from the first language of the first end user application to a language running on the application server. Typically, as described above, the client will be a thin client or a Web browser, the application running on the client will be a Web browser application or a thin client connectivity application, and the language of the client application will be Java, C, C++, or a markup language, as HTML or a derivative of HTML, such as XML

20 or Dynamic HTML or WML, or the like, and the language running on the server is PL/I. The invention facilitates transformers which convert the transaction from the first language of the end user application to a language running on the application server. After conversion, the converted transaction is processed on the application server.

25 The application processes the request and then sends the response from the application server back to the end user application. Typically, as described above, the application server will be running a PL/I based application, and the client will be a thin client written in Java or C or C++, or a Web browser, running a Web browser application or a thin client connectivity application, in a markup language, as HTML or a derivative of

30 HTML, such as XML or Dynamic HTML, or the like. The invention provides data

6

transformers which convert the response from the language or languages running on the application server or servers to the first language of the first end user application.

The end user application and the application server have at least one data transformer
5   between them. In this way, the steps of (i) converting the request from the first language of the first end user application as a source language to the language running on an application server as a target language, and (ii) converting the response from the language running on the application server, as a subsequent source language, back to the first language of the first end user application, as a subsequent target language, each comprise
10 the steps of invoking type descriptor and language metamodels of respective source and target languages, populating the metamodels with each of the respective source and target languages' data items and types, and converting the source language to the target language.

15 The end user application is, frequently, a web browser or a thin client. When the end user application is a Web browser, the end user is connected to the application server through a web server. According to a further embodiment of the invention, the web server may comprise the connector, or data transformer. The data transformer integrated with the Web server may directly convert the request, transaction, or message from a browser
20 oriented form to an application server language or to an intermediate, business or commerce oriented markup language, such as XML.

The CAM metamodel used to construct the converter comprises an invocation metamodel, an application domain interface metamodel, a language metamodel, and a
25 type descriptor metamodel. Exemplary invocation metamodel includes information chosen from the group consisting of message control information, security data, transactional semantics, trace and debug information, pre-condition and post-condition resources, and user data, etc. Exemplary application domain interface metamodel comprises information chosen from input parameter signatures, output parameter
30 signatures, and return types. Application domain interface metamodel uses one or more language metamodels, such as COBOL and PL/I metamodels.

7

The type descriptor metamodel defines physical realizations, storage mapping, data types, data structures, and realization constraints.

5   The method of the invention is applicable to situations where one of the source or target languages is object oriented, and the other of the target or source languages is not object oriented. In this situation, the language metamodel and the type descriptor metamodel together map encapsulated objects of the object oriented language into code and data of the language that is not object oriented. Additionally, the language metamodel and the

10   type descriptor metamodel maps object inheritances of the object oriented language into references and pointers in the language that is not object oriented. The method of the invention is also applicable to situations where different object oriented languages are running on different platforms, and encapsulated objects of the source language (code and data) are mapped into encapsulated objects of the target language. The method of the

15   invention is also applicable where different procedural languages are running on different platforms or applications and commands and data of the source procedural language are mapped into the target procedural language.

According to the method of the invention, there may be a plurality of applications for

20   vertical (sequential, conditional, or dependent) processing, for horizontal (parallel in time) processing, or both horizontal and vertical processing. This is to support rich transactions to and through multiple hierarchical levels and multiple parallel sequences of processing. This may be the case in business to business transactions drawing upon financial, manufacturing, scheduling, supply, and shipping databases and servers, and

25   utilizing various commercial security instruments.

A further aspect of the invention is a client-server processing system having a client, a server, and at least one transformer between the client and one or more servers,

30   A still further aspect of the invention is a processing system configured and controlled to interact with a client application. In this aspect of the invention, the system comprises, a

SVL920010036US1

server, and at least one transformer between the server and the client application, where the client has an end user application, and is controlled and configured to initiate a request with the server in a first language with a first application program and to transmit the request through a transformer to the server or servers. The server processes the

5   request in a second software application, using a second language, and returns a response to the client through a transformer.

A further aspect of the invention is a groupware system having a plurality of e-mail enabled end user applications, such as e-mail, word processing, spreadsheet, simple

10   database management (such as Lotus Approach or Microsoft Access), graphics and graphics editing, audio and audio editing, and computer-telephony integration ("CTI"), along with client level content database client services and content replication client services. Groupware integrates these e-mail enabled applications through one or more transformers and application program interfaces with transport services, directory

15   services, and storage services, including content servers and replication servers. The groupware system is configured and controlled to communicate among disparate end user applications, among disparate servers, and between disparate servers and end user applications. The groupware system comprises at least one transformer between a server and an end user application. The end user application is controlled and configured to

20   participate with a server in a first language of a first application program and the server is configured and controlled to participate with the client in a second language of a second program.

The transformer is configured and controlled to receive a request from the end user

25   application, and convert the request from the first language of the first end user application to a language running on the server. The server is configured and controlled to receive the converted request from the transformer and process the request in a second language with a second application program residing on the server, and to thereafter transmit a response through a transformer back to the end user application.

30

9

A still further embodiment of the invention is the provision of rich transaction processing. Rich transactions are nested transactions that span to, through, and/or across multiple servers. The spanning across nested servers may be horizontal, that is parallel dependent transactions, or vertical, that is, serial dependent transactions. Rich transactions may be

5   long lived, on-going transactions, or complex business-to-business transactions, especially those with multiple dependencies or contingencies, volume and prompt payment discounts, late delivery and late payment penalties, and with financial processing, such as electronic letters of credit, electronic bills of lading, electronic payment guarantees, electronic payment, escrow, security interests in the goods, and the

10   like. In a rich transaction environment, some transaction servers may be positioned as clients with respect to other transactions for certain sub transactions making up the rich transaction.

A still further embodiment of the invention is a tool, that is, a software developer's kit,

15   characterized in that the program product is a storage medium (as a tape, floppy disks, a CD-ROM, or a hard drive or hard drives on one of more computers) having invocation metamodels, application domain interface metamodels, and language metamodels, and computer instructions for building a metamodel repository of source and target language metamodels. The program product also contains computer instructions for building

20   connector stubs from the metamodels. The program product further carries computer instructions to build a transformer.

While the invention has been described in summary form as having a single level of connectors, it is, of course, to be understood that such connectors may be present at

25   various levels in the processing hierarchy, for example between Web Clients and Web servers, between web servers and application servers, between application servers and database servers, and between application servers or database servers or both and various specialized repositories.

30   It is also to be understood, that while the invention has been summarized in terms of individual clients and individual servers, there may be multiple clients, multiple servers,

SVL920010036US1

and applications that function as both clients and servers, as exemplified by groupware applications, and there might be multiple parallel lines and/or multiple hierarchical levels of application servers, data servers, and databases, as in systems for rich transactions.

5

The Figures

Various elements of the invention are illustrated in the FIGURES appended hereto.

10   FIGURE 1 illustrates a system with multiple application components, including a Netscape Internet Explorer browser, Net.Commerce on a Sun Solaris server, Oracle and DB2 on a database server, SAP running on AIX, a CICS 390 server, an IMS 390 server, DB2 and DL/I on a S/390 platform, a Windows 200 client, and Baan running on an HP Unix server.

15   FIGURE 2 illustrates the roles of message sets, SQL stored procedures, legacy applications, and programming languages as inputs to the metadata repository of the Common Application Metamodel to facilitate enterprise application integration at run time.

20   FIGURE 3 illustrates that the Common Application Metamodel of the invention consists of three kinds of metamodels, i.e., an invocation metamodel, an application-domain interface metamodel, and a language metamodel. For any given application-domain metamodel it may use one or many language metamodels, and there could be zero or many invocation metamodels.

25

FIGURE 4 illustrates an IMS OTMA metamodel, with an OTMA Invocation Metamodel, an IMS Transaction Message Metamodel application interface, which could use a COBOL Metamodel, a C Metamodel, or other language metamodels.

30   FIGURE 5 illustrates how a tool can be used to generate an XML document describing application program interface. First, an object model, i.e., a CAM metamodel, is created

11

to capture interface definitions about an application server. Then a tool reads and parses the source definitions of an application program and generates an XML document by retrieving the object model's information from a repository.

5   FIGURE 6 illustrates a development phase scenario where a Common Application Metamodel Rose file, e.g., a COBOL metamodel, a PL/I metamodel, an MFS metamodel, a BMS model, or the like is read into a toolkit, to generate a DTD and XML schema and Java code for a Rose model. A source file of an application, as a COBOL source file, a PL/I source file, an MFS source file, a BMS source file, or the like, is read into an

10   importer. The importer parses the source code and generates, as output, an XMI instance file, i.e., XML documents, by reading in the Java code of the Rose model of the application source files.

FIGURE 7 illustrates a metamodel for application interfaces, which enables integration of

15   application components into an event based messaging model, including flow models. The flow and messaging middle invokes applications through the application interface. These interfaces are access points to the applications through which all input and output is connected to the middleware. The interfaces are described in terms of the Application Interface Metamodels. Transformation processing according to he metamodel could take

20   place in source/client applications, target applications, or a gateway.

FIGURE 8 illustrates the application of the Common Application Metamodel during execution time. As shown, the CAM model facilitates connectivity between a back-end IMS application and a Web file (e.g., SOAP complaint XML documents). This is

25   accomplished by using information captured in the model to perform data transformations from one platform to another in a mixed language environment shown.

FIGURE 9 illustrates TDLang base classes, where the Type Descriptor metamodel is used as a recipe or template for runtime data transformation, with the language specific

30   metamodel for overall data structures and field names, and without duplicating the aggregation associations present in the language model.

12

SVL920010036US1

FIGURE 10 illustrates an MOF class instance at the M2 level as a Type Descriptor Metamodel.

5 FIGURE 11 illustrates the association between a TDLangElement of the TDLang Metamodel with a Platform Compiler Type of the Type Descriptor Metamodel.

FIGURE 12 illustrates various enumeration types for the Type Descriptor Metamodel, e.g., sign coding, length encoding, floating type, accessor, packed decimal sign, and

10 bitModelPad.

FIGURE 13 illustrates a PL/I language metamodel, which is usable by application programs to define data structures which represent connector interfaces. This metamodel is an MOF class instance at the M2 level.

15

FIGURE 14 illustrates the associations between the PL/I metamodel with the TDLang base classes which are the TDLangClassifier, the TDLanguageComposedType, and the TDLanguageElement for the PL/I Metamodel.

20 FIGURE 15 illustrates enumerations of Mode Values, Base Values, LengthTypes, and StringTypeValues for the PL/I Metamodel.

FIGURE 16 illustrates a simplified network configuration for a "rich" transaction where, for example, an order is entered at a terminal, and is passed to and through a Web server

25 to a manufacturer's application server. The manufacturer's application server searches through it's own database server, as well as its vendors' dissimilar and incompatible database servers and databases, transparently connected by the connectors described herein, querying for statuses, prices, and delivery dates, of components, and placing orders for needed components to satisfy the order.

30

13

FIGURE 17 illustrates a group ware session spread across multiple group ware applications running on multiple, disparate platforms, and connected using the common application metamodel described herein.

5 FIGURE 18 illustrates a commercial transaction where real goods are shipped from a seller to a buyer, and various forms of electronic payment and secured electronic payment are used by the buyer to pay the seller, with banks and financial institutions connected using the common application metamodel described herein.

Detailed Description of the Invention

10

Definitions. As used herein the following terms have the indicated meanings. "Handshaking" is the exchange of information between two applications and the resulting agreement about which languages, capabilities, and protocols to use that precedes each 15 connection.

An "application program interface" (API) is a passive specific method prescribed by a computer operating system or by another application program by which a programmer writing an application program can make requests of the operating system or another 20 application. Exemplary is SAX (Simple API for XML), an connector that allows a programmer to interpret a Web file that uses the Extensible Markup Language, that is, a Web file that describes a collection of data. SAX is an *event-driven* interface. The programmer specifies an event that may happen and, if it does, SAX gets control and handles the situation. SAX works directly with an XML parser.

25

A "connector" as used herein is a dynamic, run-time, interface between platforms that stores the functions and parameters of the target platform or program, and binds with the target platform program in real time.

30 A "stub" is a small program routine that provides static interfaces to servers. Precompiled stubs define how clients invoke corresponding services on the server. The stub

14

substitutes for a longer program on the server, and acts as a local call or a local proxy for the server object. The stub accepts the request and then forwards it (through another program) to the remote procedure. When that procedure has completed its service, it returns the results or other status to the stub which passes it back to the program that

5   made the request. Server services are defined in the stub using an Interface Definition Language ("IDL"). The client has an IDL stub for each server interface that it accesses and includes code to perform marshaling. Server stubs provide static interfaces to each service exported by the server.

10 "CICS" (Customer Information Control System) is the online transaction processing program from IBM that, together with the Common Business Oriented Language programming language, is a set of tools for building customer transaction applications in the world of large enterprise mainframe computing. Using the programming interface provided by CICS to write to customer and other records (orders, inventory figures,

15 customer data, and so forth) in a CICS, a programmer can write programs that communicate with online users and read from a database (usually referred to as "data sets") using CICS facilities rather than IBM's access methods directly. CICS ensures that transactions are completed and, if not, it can undo partly completed transactions so that the integrity of data records is maintained. CICS products are provided for OS/390,

20 UNIX, and Intel PC operating systems. CICS also allows end users to use IBM's Transaction Server to handle e-business transactions from Internet users and forward these to a mainframe server that accesses an existing CICS order and inventory database.

"IMS" (Information Management System) is the system from IBM that, together with

25 IBM's Enterprise Systems Architecture (IMS/ESA) provides a transaction manager and a hierarchical database server.

"MQ" is the MQSeries IBM software family whose components are used to tie together other software applications so that they can work together. This type of application is

30 often known as business integration software or middleware. Functionally, MQSeries provides a communication mechanism between applications on different platforms, an

15

integrator which centralizes and applies business operations rules, and a workflow manager which enables the capture, visualization, and automation of business processes. MQSeries connects different computer systems, at diverse geographical locations, using dissimilar IT infrastructures, so that a seamless operation can be run. IBM's MQSeries

5   supplies communications between applications, and between users and a set of applications on dissimilar systems. Additionally, MQSeries' messaging scheme requires the application that receives a message to confirm receipt. If no confirmation materializes, the message is resent by the MQSeries.

10   "Rose" is an object-oriented Unified Modeling Language (UML) software design tool intended for visual modeling and component construction of enterprise-level software applications. It enables a software designer to visually create (model) the framework for an application by blocking out classes with actors (stick figures), use case elements (ovals), objects (rectangles) and messages/relationships (arrows) in a sequence diagram

15   using drag-and-drop symbols. Rose documents the diagram as it is being constructed and then generates code in the designer's choice of C++, Visual Basic, Java, Oracle8, Corba or Data Definition Language.

Common Application Metamodel Overview. The Common Application Metamodel

20   (CAM) brings interconnectivity to the environment illustrated in FIGURE 1. FIGURE 1 illustrates a typical system 101 with multiple application components, including a Netscape Internet Explorer browser 103, Net.Commerce 105 on a Sun Solaris server 107, Oracle 109 and DB2 111 on a database server 113, SAP 115 running on AIX 117, a CICS 390 server 119, an IMS 390 server 121, DB2 123 and DL/I 125 on a S/390 platform 127,

25   a Windows 2000 client 129, and Baan 131 running on an HP Unix server 133. The Common Application Metamodel (CAM) is metadata interchange method, tool, and system for marshaling and applying information needed for accessing enterprise applications, such as in FIGURE 1, in a source language and converting them to a target language. CAM consists of language metamodels and application domain interface

30   metamodels, as shown in FIGURE 2, which illustrates the roles of message sets 203, SQL stored procedures 205, legacy applications 207, and programming languages 209 as

16

inputs to the metadata repository 211 of the Common Application Metamodel to facilitate enterprise application integration 221.

Exemplary metamodels include C, C++, Java, COBOL, PL/I, HL Assembler, IMS
5   transaction messages, IMS MFS, CICS BMS, and MQSeries messages models, as shown in FIGURE 3, which illustrates the Common Application Metamodel of the invention, with an invocation metamodel 301, an application-domain interface metamodel 303, and a language metamodel 305.

10   FIGURE 4 illustrates an IMS OTMA application interface metamodel 411, with an OTMA Invocation Metamodel 421, an IMS Transaction Message Metamodel 423, a COBOL Metamodel 425, and a C Metamodel 427.

FIGURE 5 illustrates the flow of information from an existing application 501, through
15   an interface 503 to an object model containing application interface metadata. This application interface metamodel is stored in the metadata repository 505, and, at an appropriate time, retrieved from the metadata repository 505, combined with a source program 507 in a generation tool 509, and used to generate a target file 511, as an XML file, i.e., an XMI instance file. CAM is highly reusable and independent of any particular
20   tool or middleware.

Development Stage. With CAM, tooling can now easily provide solutions to access enterprise applications, e.g. IMS applications. By parsing each source file and generating XML documents based on the CAM model, COBOL copybook, PL/I copybook, MFS
25   Source, BMS Source, etc., tools can provide connector solutions to IMS, and CICS, etc.

In this regard, FIGURE 6 illustrates a development phase scenario where a Common Application Metamodel Rose file 601, e.g., a COBOL metamodel, a PL/I metamodel, an MFS metamodel, a BMS model, or the like is read into a toolkit 603, to generate a DTD
30   and schema for a Rose model and Java code for a Rose model 605. A source file of an application 607, as a COBOL source file, a PL/I source file, an MFS source file, a BMS

source file, or the like, and the Java code for the Rose model 609 are read into an

Importer 611. The importer parses the source code and provides, as output, an XMI

instance file 613, i.e., XML documents, of the application source files.

5   FIGURE 7 shows a CAM metamodel for application interfaces. This Figure depicts a

runtime connector 701 with invocation and transformation capabilities, interfacing with

an existing application program 703 through an interface 705 containing the existing

application program's interface definition, in accordance with the application interface

metamodel 707. The Application Interface metadata is stored in a metadata repository

10   709.

The flow and messaging middleware 713 invokes applications 703 through the

application interfaces 705. These interfaces 705 are the access points to the applications

703 through which all input and output is connected to the middleware 713. The

15   interfaces 705 are described in terms of the Application Interface Metamodel.

Transformation processing according to the metamodel could take place in source/client

applications, target applications, or a gateway.

Because CAM also provides physical representation of data types and storage mapping to

20   support data transformation in an enterprise application integration environment, it

enables Web services for enterprise applications.

At development time CAM captures information that facilitates:

        a). connector and/or connector-builder tools,

        b). data driven impact analysis for application productivity and quality assurance, and

25   c). viewing of programming language data declarations by developers.

The CAM metamodel files are inputs to toolkits used to generate DTD files, XML

schemas, and Java classes which represent the CAM model. Importers parse each source

file (e.g. COBOL or PL/I copybook, MFS source, and BMS, etc.), and then generate

30   XML documents (i.e. XML instance files) based on Java classes generated by the

XMI/MOF2 toolkit.

18

Run Time.   At run time CAM provides information which facilitates transformation in an enterprise application integration environment where it provides data type mapping between mixed languages, facilitates data translations from one language and platform

5   domain into another.

FIGURE 8 illustrates the application of the Common Application Metamodel during run time. As shown, SOAP compliant XML documents 803 are received in, for example, IBM WebSphere middleware, 805, which contains an IMSConnector for Java 807, and is

10   in contact with an XML Repository 809, containing the XMI instance files for the CAM model. The IBM WebSphere middleware sends the transformed file to the IMS system 811, which contains an instance of IMS Connect 813 and the IMS transactional application program 815. CAM facilitates connectivity between the back-end IMS application 815 and the Web file (e.g., SOAP compliant XML documents) 803. The

15   CAM accomplishes this by using CAM model information (from repository 809) to perform data transformations from one platform to another in the mixed language environment shown.

Type Descriptor Metamodel. One important feature provided by CAM is the Type

20   Descriptor metamodel. The Type Descriptor metamodel defines the physical realization, storage mapping, and the constraints on the realization (such as justification). This metamodel provides a physical representation of individual fields of a given data structure. When supporting data transformation in an enterprise application integration environment, the model provides data type mapping between mixed languages. It also

25   facilitates data translations from one language and platform domain into another. The metamodel is used for runtime data transformation (or marshaling) with a language-specific metamodel for overall data structures and field names.

1.   Common Application Metamodel for Application Interfaces

30

SVL920010036US1

The interconnection of disparate and dissimilar applications running on different software platforms, as shown in FIGURE 1, with different operating systems, physical platforms, and physical realizations is accomplished through connectors that incorporate the interconnection metadata. Connectors are a central part of the application framework for

5   e-business. The end user demand is to connect to anything interesting as quickly, and as easily, as possible.

A connector is required to match the interface requirements of the adapter and the legacy application. It is also required to map between the two interfaces. Standardized

10   metamodels for application interfaces presented herein allow reuse of information in multiple connector tools. These standardized metamodels not only reduce work to create a connector, but also reduce work needed to develop connector builder tools.

The connectors built using the common application metamodel of our invention provide

15   interoperability with existing applications. The connectors support leveraging and reuse of data and business logic held within existing application systems. The job of a connector is to connect from one application system server "interface" to another. Therefore, an application-domain interface metamodel describes signatures for input and output parameters and return types for a given application system domain (e.g. IMS,

20   MQSeries); it is not for a particular IMS or MQSeries application program. The metamodel contains both syntactic and semantic interface metadata.

1.   a.   End-to-End Connector Usage Using Common Application Metamodel

25   The Common Application Metamodel (CAM) consists of meta-definitions of message signatures, independent of any particular tool or middleware. Different connector builder tools can use this information to ensure the "handshaking" between these application programs, across different tools, languages, and middleware. For example, if you have to invoke a MQSeries application, you would need to build a MQ message using data from

30   a GUI tool and deliver it using the MQ API. Similarly, when you receive a message from the MQSeries application, you would need to get the buffer from MQSeries, parse it and

then put it into a GUI tool data structure. These functions can be designed and implemented efficiently by a connector builder tool using CAM as standardized metamodels for application interfaces.

5   CAM can be populated from many sources, including copy books, to generate HTML forms and JavaServer Page (JSP) for gathering inputs and returning outputs. An example of a connector as depicted in the previous figure is that the flow and message middleware makes a function call to an enterprise application by calling the connector which then calls the enterprise application API. The connector does language and data type

10   mappings, for example, to translate between XML documents and COBOL input and output data structures based on CAM. Connectors and CAM provide the end-to-end integration between the middleware and the enterprise applications.

Using IMS as an example. Let's say that you must pass an account number to an IMS

15   transaction application program from your desktop to withdraw $50.00. With CAM and a connector builder tool, you will first generate an input HTML form and an output JSP; and develop a middleware code necessary to support the request. The desktop application fills the request data structure (i.e. an input HTML form) with values and calls the middleware. The middleware service code will take the data from the GUI tool, build an

20   IMS Connect XML-formatted message, and deliver the message to the IMS gateway (i.e. IMS Connect) via TCP/IP. IMS Connect translates between the XML documents and the IMS message data structures in COBOL using the metadata definitions captured in CAM. It then in turn sends the IMS message data structures to IMS via Open Transaction Manager Access (OTMA). The IMS COBOL application program runs, and returns the

25   output message back to the middleware service code via IMS Connect. The middleware service code gets the message and populates the output JSP page (i.e. previously generated GUI tool reply data structures) with the reply data. The transaction output data will then be presented to the user.

30   **2.   Common Application Metamodel**

SVL920010036US1

CAM is used to describe information needed to easily integrate applications developed in common programming models with other systems. The CAM metamodel can be used for both synchronous and asynchronous invocations.

5   **2.   a.   Common Application Metamodel**

The common application metamodel depicted as follows consists of an invocation metamodel and an application-domain interface metamodel which uses language metamodels. For any given application-domain interface metamodel, it may use one or

10   many language metamodels, but, there could be zero or more invocation metamodels.

The common connector metamodel is illustrated in FIGURE 3. It has an Invocation Metamodel 301, an Application-Domain Interface Metamodel 303, and a Language Metamodel 305.

15

2.   a.   i.   Invocation Metamodel

The invocation metamodel 301 consists of one or more of the following possible metadata elements. However, for a particular invocation, it could include only one or

20   many of the following metadata elements.

Message-control information. This includes message control information, such as the message connection name, message type, sequence numbers (if any), and various flags and indicators for response, commit-confirmation, and processing options by which a

25   client or server can control message processing to be synchronous or asynchronous, etc.

The connection name can be used by the application system server to associate all input and output with a particular client. The message type specifies that the message is a response message; or that commit is complete. It can also indicate server output data to

30   the client, or client input data to the server.

22

Security data -- This includes authentication mechanism, and security data, e.g. digital certificates, identity, user name and password, etc. It may also include authorization information to indicate whether the data can be authorized via a role based or ACL (access control list) based authorization.

5

Transactional semantics -- This will carry transaction information, e.g. local vs. global transaction; two-phase commit vs. one-phase commit, and transaction context, etc.

Trace and debug -- Trace and debugging information are specified as part of the

10   metamodel.

Precondition and post-condition resource -- This describes application state precondition and post-condition relationships.

15   User data -- This includes any special information required by the client. It can contain any data.

2.   a.   ii.   Application-Domain Interface Metamodel

20   The application-domain interface metamodel 303, as discussed earlier, describes signatures for input and output parameters and return types for application system domains.

2.   a,   iii.   Language Metamodel

25

The language metamodel 305, e.g. COBOL metamodel, is used by enterprise application programs to define data structures (semantics) which represent connector interfaces. It is important to connector tools to show a connector developer the source language, the target language, and the mapping between the two. The CAM language metamodel also

30   includes the declaration text in the model which is not editable (i.e. read-only model). Because the connector/adapter developer would probably prefer to see the entire COBOL

23

data declaration, including comments and any other documentation that would help him/her understand the business role played by each field in the declaration.

The language metamodel is also to support data driven impact analysis for application

5   productivity and quality assurance. (But, it is not the intention of the CAM to support reproduction of copybooks.)

The language metamodels describing connector data are listed as follows:

10   * C

* C++

* COBOL

* PL/I

15   **2.   a.   iv.   Type Descriptor Metamodel**

The Type Descriptor metamodel is language neutral and defines the physical realization, storage mapping and the constraints on the realization such as justification. This metamodel provides physical representation of individual fields of a given data structure.

20   The type descriptor metamodel is to support data transformation in an enterprise application integration environment to provide data types mapping between mix languages. It also facilitates data translations from one language and platform domain into another. This metamodel will be used as a recipe for runtime data transformation (or marshaling) with language specific metamodel for overall data structures and fields

25   names.

3.   An Example of Common Connector Metamodel

IMS OTMA (Open Transaction Manager Access) is a transaction-based, connectionless

30   client/server protocol within an OS/390 sysplex environment. An IMS OTMA transaction message consists of an OTMA prefix, plus message segments for input and output

24

requests. Both input and output message segments contain llzz (i.e. length of the segment and reserved field), and application data. Only the very first input message segment will contain transaction code in front of the application data. IMS transaction application programs can be written in a variety of languages, e.g. COBOL, PL/I, C, and Java, etc.

5 Therefore, the application data can be in any one of these languages.

As shown in FIGURE 4, an IMS OTMA connector metamodel 401 is composed of an invocation metamodel 403 and an IMS transaction message metamodel 405, as well as a COBOL metamodel 407 and a C metamodel 409. As depicted in FIGURE 4, the

10 invocation metamodel 401 is the OTMA prefix, and the IMS transaction message metamodel 405 is the application-domain interface metamodel for the IMS application system which uses language metamodels. Metamodels for COBOL 407 and C 409 are shown.

15 **4.   Type Descriptor Metamodel**

The type descriptor metamodel presents a language and platform independent way of describing implementation types, including arrays and structured types. This information is needed for marshaling and for connectors, which have to transform data from one

20 language and platform domain into another. Inspections of the type model for different languages can determine the conformance possibilities for the language types. For example, a long type in Java is often identical to a binary type (computational-5) in COBOL, and if so, the types may be inter-converted without side effect. On the other hand, an alphanumeric type in COBOL is fixed in size and if mapped to a Java type, loses

25 this property. When converted back from Java to COBOL, the COBOL truncation rules may not apply, resulting in computation anomalies. In addition, tools that mix languages in a server environment (e.g., Java and COBOL in CICS and IMS) should find it useful as a way to determine how faithfully one language can represent the types of another.

30 Therefore, an instance of the type descriptor metamodel describes the physical representation of a specific data type for a particular platform and compiler.

SVL920010036US1

4.   a.   TDLang Metamodel

The TDLang metamodel serves as base classes to CAM language metamodels by

5   providing a layer of abstraction between the Type Descriptor metamodel and any CAM

language metamodel. All TDLang classes are abstract and common to all the CAM

language metamodels. All associations between TDLang classes are marked as "volatile,"

"transient," or "derived" to reflect that the association is derived from the language

metamodel.   The TDLang model does not provide any function on its own, but it is the

10   type target for the association from the Type Descriptor metamodel to the language

metamodels.

FIGURE 9 illustrates the structure of the TDLang Metamodel, with the TDLangClassifier

501, the TDLangComposedType 503 and the TDLangElement 505.

15

With the TDLang base classes, the Type Descriptor metamodel can be used as a recipe

for runtime data transformation (or marshaling) with the language-specific metamodel for

overall data structures and field names, without duplicating the aggregation associations

present in the language model.

20

4.   b.   Type Descriptor Metamodel

This metamodel is a MOF Class instance at the M2 level. FIGURE 10 shows the

relationships within the type descriptor Metamodel, including the PlatformCompilerType

25   601, the InstanceTDBase 603, the ArrayTD 605, the AggregateInstanceTD 607, the

Simple InstanceTD 609, and the InstanceType 611. The InstanceType 611 comprises

definitions of the StringTD 613, the AddressTD 615, the NumberTD 617, and the

FloatTD 619. FIGURE 11 illustrates a higher level view of the TDLanguageElement and

the PlatformCompilerType 601. FIGURE 12 illustrates enumerations of signCoding

30   801, lengthEncoding 803, floatType 805, accessor 807, packedDecimalSign 809, and

bitModePad 811.

SVL920010036US1

4.   c.   Type Descriptor and Language models

The Type Descriptor model is attached to the CAM Language model by a navigable

5   association between TDLangElement and InstanceTDBase. TDLangElement is the base

language model type used to represent a declared data item, i.e., an instance of a type.

InstanceTDBase is the base Type Descriptor model type used to represent the

implementation-specific instance of this same declared data item. InstanceTDBase is

abstract; only one of its subtypes may be instantiated.

10

It is possible that a data item declared in a programming language may have different

implementations. These differences are induced by hardware platform, system platform,

and compiler differences. This possibility is modeled by the PlatformCompilerType

model type. The association between TDLangElement and PlatformCompilerType is

15   many to one, and the association between PlatformCompilerType and InstanceTDBase is

one to one. To navigate from the language model, it is necessary to know what

PlatformCompilerType is to be assumed. It is possible that an implementation, upon

importing a model instance, will wish to remove from the model the

PlatformCompilerType instances that are not of interest.

20

The association between TDLangElement and InstanceTDBase is modeled in this manner

to allow for extending the model to include an association between

PlatformCompilerType and a new type that more fully describes the hardware platform,

the system platform, and the compiler.

25

Data element instances may be defined as repeating groups or arrays. This is modeled as

a one to many association between InstanceTDBase and the ArrayTD model type. There

would be one ArrayTD instance in this association for each dimension, subscript, or

independent index of the data element. These instances hold information about the

30   bounds and accessing computations.

SVL920010036US1

The association is ordered in the same order as the corresponding association in the language model, and reflects the syntactic ordering of the indices as defined by the programming language. The rationale for this choice is the resulting equivalence of navigation and processing algorithms between the language model and the Type

5   Descriptor model. Another choice, perhaps more advantageous to marshaling engines, would be to have the ordering of the indices from the smallest stride to the largest. This allows a marshaling engine to process the array in its natural storage order, assuming it is laid out in the usual contiguous fashion. A marshaling engine can compute this order by re-sorting the association targets according to the stride formulas if desired.

10

Array information may be a complex property of the data element or of its type, and various languages and programming practices seem to fall on either side. The typedef facility of C and C++ allows the definition of some array types from typedefs, but only where the array definitions are applied to the topmost elements of typedef aggregates. For

15   example, consider the following typedef:

```
typedef struct {
        int A;
        struct {
20              int C;
                char D;
                struct {
                        int F;
                        int G;
25              } E;
        } B;
} X;
```

This typedef can be used to create a new typedef for a fixed size array, e.g.

30

```
typedef X Q[10];
```

But it is not possible to create a new typedef from X that makes any of the subcomponents of X, e.g., D or E, into an array. This example and many others point out

35   the unclear status of array definitions in typed languages.

28

An InstanceTDBase type has two concrete subtypes, SimpleInstanceTD and AggregateInstanceTD. SimpleInstanceTD models data elements without subcomponents, while AggregateInstanceTD models data elements with subcomponents. To find the subcomponents of an AggregateInstanceTD, one must navigate back to the corresponding data element declaration in the CAM language model. There, the association between an aggregate type and its subcomponents may be navigated, leading to a set of subcomponent data elements, each of which has one or more corresponding instances in the Type Descriptor model. This introduces some model navigation complexity, but avoids duplicating the aggregation hierarchy in both the language and the Type Descriptor models. The additional processing complexity of traversal is not great, and considerable simplification is obtained in algorithms that would modify the model to add, delete or rearrange subcomponents in an aggregation.

A SimpleInstanceTD model type is also associated one to one with a BaseTD model type. The BaseTD model type is specialized to hold implementation information that is common for all data elements of the same language type. The information that describes a 32-bit signed binary integer on a specific hardware/software platform is thus instantiated only once in a given model instantiation, no matter how many data elements may be declared with this type.

One may contemplate an association between TDLangClassifier and BaseTD matching the association between TDLangElement and InstanceTDBase. However, this is problematic in that constructions that the language regards as simple types (e.g., strings) may not map directly to simple hardware/software types. Rather than introduce more mechanisms into the Type Descriptor model to describe string implementations, a specialization of BaseTD is utilized which describes the common string implementations. Various attributes in the TypeDescriptor model are suffixed with the string "formula." These attributes contain information that may in some cases be impossible to compute without access to data created only at run-time. An example is the current upper bound of a variable-sized array or the offset to an element that follows another element whose size

29

is only known at run-time. Such information could be included as values in a model instance, but this would require a model instance for each run-time instance, and would mean that the model could only be constructed at run-time, requiring the model definition to include factories and other apparatus to create model instances at run-time. A model

5   that can be constructed from platform and compiler knowledge is much more useful, and the formulas provide a way to define concrete values when the run-time information is available. These formulas may be interpreted by marshaling engines, or they may be used to generate marshaling code, which is loaded and executed by the marshaling engine on demand.

10

4.   d.   Formulas

As used in connection with formulas, "field" refers to a component of a language data structure described by the Type Descriptor model, while "attribute" denotes part of the

15   model, and has a value representing a "property" of the field. Thus the value of a field means a run-time value in a particular instance of a language data structure, whereas the value of an attribute is part of the description of a field in a language data structure, applies to all instances of the data structure, and is determined when the data structure is modeled.

20

For most attributes in an instance of the Type Descriptor model, the value of the attribute is known when the instance is built, because the properties of the fields being described, such as size and offset within the data structure, are invariant. But if a field in a data structure is defined using the COBOL OCCURS DEPENDING ON construct or the PL/I

25   Refer construct, then some properties of the field (and properties of other fields that depend on that field's value) cannot be determined when the model instance is built.

Properties that can be defined using these language constructs are string lengths and array bounds. A property that could indirectly depend on these language constructs is the offset

30   of a field within a structure, if the field follows a variable-size field.

In order to handle these language constructs, properties of a field that could depend on these constructs (and thus the values of the corresponding attributes), are defined with strings that specify a formula that can be evaluated when the model is used.

5   However, if a property of a field is known when the model instance is built, then the attribute formula simply specifies an integer value. For example, if a string has length 17, then the formula for its length is "17".

The formulas mentioned above are limited to the following:

10   * Unsigned integers

     * The following arithmetic integer functions

$$
\begin{array}{lll}
neg(x) & := -x & \text{// prefix negate} \\
add(x,y) & := x+y & \text{// infix add} \\
sub(x,y) & := x-y & \text{// infix subtract} \\
mpy(x,y) & := x*y & \text{// infix multiply} \\
div(x,y) & := x/y & \text{// infix divide} \\
max(x,y) & := max(x,y) & \\
min(x,y) & := min(x,y) & \\
mod(x,y) & := x \bmod y &
\end{array}
$$

20

The mod function is defined as $mod(x,y) = r$ where r is the smallest non-negative integer such that x-r is evenly divisible by y. So $mod(7,4)$ is 3, but $mod(-7,4)$ is 1. If y is a power of 2, then $mod(x,y)$ is equal to the bitwise-and of x and y-1.

     * The val function

25

The val function returns the value of a field described by the model. The val function takes one or more arguments, and the first argument refers to the level-1 data structure containing the field, and must be either:

     * the name of a level-1 data structure in the language model

30   * the integer 1, indicating the level-1 parent of the variable-size field. In this case, the variable-size field and the field that specifies its size are in the same data structure, and so have a common level-1 parent.

31

The subsequent arguments are integers that the specify the ordinal number within its substructure of the (sub)field that should be dereferenced.

By default, COBOL data fields within a structure are not aligned on type-specific
5 boundaries in storage. For example, the "natural" alignment for a four-byte integer is a full-word storage boundary. Such alignment can be specified by using the SYNCHRONIZED clause on the declaration. Otherwise, data fields start immediately after the end of the preceding field in the structure. Since COBOL does not have bit data, fields always start on a whole byte boundary.

10

For PL/I, the situation is more complicated. Alignment is controlled by the Aligned and Unaligned declaration attributes. By contrast with COBOL, most types of data, notably binary or floating-point numbers, are aligned on their natural boundaries by default.

15 **4.   d.   i)   Formula examples   PL/I**

1.   Given the following structure

```
dcl                         /* offset              */
     1 c unaligned               /* "0"                   */
     ,2 c1             /*   "0"          */
20       ,3 c2   fixed bin(31)    /*   "0"              */
         ,3 c3   fixed bin(31)    /*   "4"              */
     ,2 c4             /*  "8"           */
         ,3 c5   fixed bin(31)    /*   "0"              */
         ,3 c6   fixed bin(31)    /*   "4"              */
25       ,3 c7   fixed bin(31)    /*   "8"              */
     ,2 c8   fixed bin(31)    /*  "20"               */
     ,2 c9    char( * refer(c7) ) /*  "24"                  */
     ,2 c10   char(6)          /*  "add(24,val(1,2,3))"     */
     ,2 c11   char(4)          /*  "add(add(24,val(1,2,3)),6)" */
30   ;
```

The offset of c3 would be given by the simple formula "4", but the offset of c10 would be given by the formula:

35   *"add(24,val(1,2,3))"*

32

The first argument in the above val function is 1, which indicates the current structure, c. The subsequent arguments are 2 and 3, indicating that the third element, c7, of the second level-2 field, c4, is the field to be dereferenced.

5

The offset of c11 is equal to the offset of c10 plus the length of c10 and would be given by the following formula:

$$"add(add(24,val(1,2,3)),6)"$$

10   PL/I structure mapping is not top-down, and this can be illustrated by examining the mapping of the following structure:

```
dcl                    /* offset                */
  1 a based,             /* "0"                    */
    2 b,              /*  "0"                    */
      3 b1 fixed bin(15),   /*    "0"                  */
      3 b2 fixed bin(15),   /*    "2"                  */
      3 b3 fixed bin(31),   /*    "4"                  */
    2 c,            /*  "add(8,mod(neg(val(1,1,1)),4))" */
      3 c1 char( n refer(b1)), /*    "0"                  */
      3 c2 fixed bin(31);   /*    "val(1,1,1)"              */
```

The value of b1 is given by val(1,1,1), and in order to put c2 on a 4-byte boundary, PL/I puts any needed padding before c (yes, not between c1 and c2), and hence the offset of c would be given by the following formula:

25    $$"add(8,mod(neg(val(1,1,1)),4))"$$

So if b1 contains the value 3, then this formula becomes add(8,mod(neg(3),4)), which evaluates to 9. I.e., there is one byte of padding between the structure b and the structure c.

30

The model also uses these formulas to specify the bounds and strides in an array, where the stride is defined as the distance between two successive elements in an array.

33

For example, in the following structure, the second dimension of a.e has a stride specified by the formula "4", and the first dimension by the formula "20":

```
dcl
    1 a,                    /* offset = "0"      */
5       2 b(4) fixed bin(31),   /* offset = "0"      */
                            /* lbound(1) = "1"    */
                            /* hbound(1) = "4"    */
                            /* stride(1) = "4"    */
        2 c(4) fixed bin(31),   /* offset = "16"     */
10                          /* lbound(1) = "1"    */
                            /* hbound(1) = "4"    */
                            /* stride(1) = "4"    */
        2 d(4) char(7) varying,   /* offset = "32"    */
                            /* lbound(1) = "1"    */
15                          /* hbound(1) = "4"    */
                            /* stride(1) = "9"    */
        2 e(4,5) fixed bin(31);   /* offset = "68"   */
                            /* lbound(1) = "1"    */
                            /* hbound(1) = "4"    */
20                          /* stride(1) = "20"   */
                            /* lbound(2) = "1"    */
                            /* hbound(2) = "5"    */
                            /* stride(1) = "4"    */
```

25 This means that to locate the element a.e(m,n), one would take the address of a.e and add to it $(m-1)*20 + (n-1)*4$.

If the example were changed slightly to:

```
30      dcl
            1 a(4),             /* offset = "0"      */
                            /* lbound(1) = "1"    */
                            /* hbound(1) = "4"    */
                            /* stride(1) = "40"   */
35          2 b fixed bin(31),   /* offset = "0"     */
            2 c fixed bin(31),   /* offset = "4"     */
            2 d char(7) varying,   /* offset = "8"   */
            2 e(5) fixed bin(31);   /* offset = "20"  */
                            /* lbound(1) = "1"    */
40                          /* hbound(1) = "5"    */
                            /* stride(1) = "4"    */
```

SVL920010036US1

then there is padding between d and e, but the user of the type descriptor can be blissfully unaware and simply use the stride and offset formulas to locate any given array element.

The stride for a is "40", the stride for e is "4", and the offset for e is "20". This means that

5   to locate the element a(m).e(n), one would take the address of a and add to it *(m-1)*40 + 20 + (n-1)*4.*

Finally, if the example were changed again to:

```
       dcl
10       1 a(4),              /* offset = "0"        */
                             /* lbound(1) = "1"     */
                             /* hbound(1) = "4"     */
                             /* stride(1) = "40"    */
           2 b fixed bin(31),     /* offset = "0"        */
15         2 c(8) bit(4),         /* offset = "4"        */
                             /* lbound(1) = "1"     */
                             /* hbound(1) = "8"     */
                             /* stride(1) = "4"     */
           2 d char(7) varying,    /* offset = "8"       */
20         2 e(5) fixed bin(31);   /* offset = "20"      */
                             /* lbound(1) = "1"     */
                             /* hbound(1) = "5"     */
                             /* stride(1) = "4"     */
```

25   then the computations for a.e are the same as above, but the computations for a.c become interesting.

The stride for a is still "40", the stride for c is "4" (but this "4" is a count of bits, not bytes), and the byte offset for c is "4". To locate the element a(m).c(n), one needs both a

30   byte address and a bit offset. For the byte address, one would take the address of a and add to it (m-1)*40 + 4 + ((n-1)*4)/8. The bit offset of a(m).c(n) would be given by mod((n-1)*4,8).

4.   e.   Type Descriptor Specification

35

4.   e.   i.   TDLang Metamodel Specification

35

TDLang Classes - General Overview.   TDLang classes serve as a layer of abstraction between any CAM language model and the TypeDescriptor model.

5   Since any CAM language model can plug into the TDLang model, the Type Descriptor model only needs to understand how to interface with TDLang in order to access any CAM language model.

The TDLang model does not provide any function on its own and therefore only makes

10   sense when it is attached to a language model. TDLang is common to all the CAM language models and is the type target for the association from TypeDescriptors to the language models.

Note all TDLang classes are abstract and they serve as the base classes to the language

15   metamodels.

TDLangClassifier.   TDLangClassifier is the parent class of all language-specific Classifier classes and TDLangComposedType. The TDLangSharedType association is derived from the language's "+sharedType" association from Element to Classifer class.

20   The association should be marked "volatile," "transient," or "derived" to reflect that the association is derived from the language model. The TDLangClassifier is derived from TDLangModelElement

TDLangElement.   TDLangElement is the parent class of all language-specific Element

25   classes.   The tdLangTypedElement association is derived from the language's "+typedElement" association from Classifer to Element class.   The association should be marked "volatile", "transient", and "derived" to reflect that the association is derived from the language model.

30   The tdLangElement association is derived from the language's "+element" association from Classifer to Element class.   The association should be marked "volatile,"

36

"transient," or "derived" to reflect that the association is derived from the language model.

TDLangComposedType. The TDLangComposedType is the parent class of all
language-specific ComposedTypes. The TDLangGroup association is derived from the
language's "+group" association from Element to ComposedType class. The association
should be marked "volatile," "transient," or "derived" to reflect that the association is
derived from the language model. The TDLangComposedType is derived from
TDLangClassifier.

4.   e.   ii.   Type Descriptor Metamodel Specification

The Type Descriptor package defines a model for describing the physical implementation
of a data item type. This model is language neutral and can be used to describe the types
of many languages. Inspections of the type model for different languages can determine
the conformance possibilities for the language types. For example, a long type in Java is
often identical to a binary type in COBOL, and if so, the types may be interconverted
without side effect. On the other hand, an alphanumeric type in COBOL is fixed in size
and if mapped to a Java type, will lose this property. When converted back from Java to
COBOL, the COBOL truncation rules may not apply, resulting in computation
anomalies.

AggregateInstanceTD. For each instance of an aggregate, there is an instance of this
class. To find the children of this aggregate, one must navigate the associations back to
language Classifier then downcast to language Composed Type and follow the
association to its children.

    Derived from InstanceTDBase

    Public Attributes:

        union : boolean = false

    Distinguishes whether the aggregation is inclusive (e.g. a structure) or exclusive
(e.g. a union).

If union = true, storage might be overlaid and as a result the interpretation of the content may be uncertain.

5   ArrayTD.   ArrayTD holds information for array types.

Public Attributes:

arraryAlign : int

Alignment requirements in addressible units for each element in the array.

strideFormula : string

10   A formula that can be used to calculate the displacement address of any element in the array, given an index.

strideInBit : boolean

True indicates strideFormula value in bits

15   False indicates strideFormula value in bytes

upperBoundFormula : String

Declared as a String for instances when this value is referenced by a variable.

This attribute supplies the upperbound value of a variable size array

Upperbound is required when traversing back up the entire structure.

20   lowerBoundFormula : String

Declared as a String for instances when this value is referenced by a variable.

This attribute supplies the lowerbound value of a variable size array.

InstanceTDBase.   InstanceTD has instances for each declared variable and structure

25   element.

To find the parent of any instance (if it has one) one must navigate the associations back to TDLangElement, follow the association to TDLangClassifier to locate the parent, then follow the associations back into the TypeDescriptor model.

Public Attributes:

30   offsetFormula : string

A formula for calculating the offset to the start of this instance.

38

This attribute is String because this field may not always be an integer value. For example, (value(n) + 4) could be a possible value.

5   NOTE: The offset value is calculated from the top-most parent. (e.g., for a binary tree A-->B, A-->C, B-->D, B-->E. The offset to D is calculated from A to D, not B to D)

contentSizeFormula : string

Formula for calculating the current size of the contents

10   allocSizeFormula : string

Formula for calculating the allocated size of the instance

formulaInBit : boolean

True indicates offsetFormula, contentSizeFormula, and allocSizeFormula values are in bits

15   False indicates offsetFormula, contentSizeFormula, and allocSizeFormula values are in bytes

defaultEncoding : String

Physical encoding - how many bits used to encode code points, how are the code points mapped onto bit patterns

20   Contains info on how string content is encoded: EBCDIC, ASCII, UNICODE, UTF-8, UTF-16, codepage numbers, etc...

accessor : enumeration

Specifies access-rights for this element.

25   defaultBigEndian : boolean

True if this element is Big Endian format.

floatType : enumeration

Specifies this element's float type.

30   PlatformCompilerType. A specific data type for a particular platform and compiler.

39

NOTE: There needs to be some way to identify the platform and compiler. This class can be specializedor have an attribute, or be simplified by putting an attribute on InstanceTDBase.

Public Attributes:

5   platformCompilerType : String

This attribute specifies the type of compiler used to create the data in the language model. Usage of this string is as follows:

"Vendor name, language, OS, hardware platform" (e.g., "IBM, COBOL, OS390, 390" or "IBM, PLI, WinNT, Intel")

10

SimpleInstanceTD. An instance of a Simple type in the language model.

Derived from InstanceTDBase

NumberTD

15   All numbers representations.

Currently includes Integers and Packed Decimals

Note required fields for these types of Numbers:

20   *Integer*

Base=2

MSBU=0 or 1

Signed/Unsigned

Size (in bytes) = 1,2,4,8 (16)

25

Packed Decimal

Base=10

MSBU=0

Signed

30   Width=1-63

Float

Base=2(IEEE), 16(Hex)

MSBU=0 or 1

Signed

5   Size (in bytes)=4,8,16

Encoding Details...

Derived from BaseTD

10   Public Attributes:

base : int

The base representation of this number.   2=binary, 10=decimal, 16=hex,

...

baseWidth : int

15   Number of bits used to represent base:

e.g. binary = 1, decimal = 8, packed = 4

baseInAddr : int

Number of baseWidth units in addressable storage units - e.g. decimal =

1, packed = 2, binary = 8 where the addressable unit is a byte. If the

20   addressable unit was a 32 bit word, decimal would be 4, packed would be

8, and binary would be 32.

baseUnits : int

Number of base units in the number. This times the base width must be

less than or equal to the width times the addrUnit.

25   For example, a 24 bit address right aligned in a 32 bit word would have

base = 1, basewidth=24, baseInAddr=8, width=4.

signCoding : enumeration

A set of enumerations - 2's complement, 1's complement, and sign

magnitude for binary; zone signs for decimal, packed signs for packed

30   decimal, and unsigned binary, unsigned decimal.

checkValidity : boolean

41

True if language model is required for picture string support

packedDecimalSign : enumeration

Used to determine the code point of the sign in COBOL decimal numbers,

where the sign is combined with the leading or trailing numeric digit.

5

FloatTD

Floating points

Derived from BaseTD

10

Public Attributes:

floatType : enumeration

Specifies this element's float type.

1 StringTD

15

Alphanumeric type

Derived from BaseTD

Public Attributes:

encoding : String

Physical encoding - how many bits used to encode code points, how are

20

the code points mapped onto bit patterns

Contains info on how string content is encoded: EBCDIC, ASCII,

UNICODE, UTF-8, UTF-16, codepage numbers, etc...

lengthEncoding : enumeration

25

Possible values for lengthEncoding:

- Fixed length (where total length equals declared string length)

- Length prefixed (where total length equals declared string length plus

length of header bytes; usually 1, 2, or 4 bytes)

- Null terminated (known as varyingZ strings in PL/I) (where a null

symbol is added to the end of string; so the maximum length could be up

to declared string length plus one byte to represent null character)

maxLengthFormula : String

5   Formula specifying the maximum length of this string.

checkValidity : boolean

True if language model is required for picture string support

textType : String = Implicit

Value is 'Implicit' or 'Visual'

10   orientation : StringTD = LTR

Value is 'LTR', 'RTL', 'Contextual LTR', or 'Contextual RTL'

where LTR = Left to Right

and RTL = Right to Left

Symmetric : boolean = true

15   True if symmetric swapping is allowed

numeralShapes : String = Nominal

Value is 'Nominal', 'National', or 'Contextual'

textShape : String = Nominal

Value is 'Nominal', 'Shaped', 'Initial', 'Middle', 'Final', or 'Isolated'

20

AddressTD

Type to represent pointers/addresses

Rationale for this class:

Addresses should be considered separate from NumberTD class because some languages

25   on certain machines (e.g., IBM 400) represent addresses with additional information,

such as permission type (which is not represented in NumberTD class)

Derived from BaseTD

Public Attributes:

30   permission : String

Specifies the permission for this address. Used primarily for AS/400 systems.

43

bitModePad : enumeration

Specifies the number of bits for this address. Used to calculate padding.

BaseTD

5   The base class of typeDescriptor.

The BaseTD model type is specialized to hold implementation

information which is common for all data elements of the same language

type. The information which describes a 32 bit signed binary integer on a

10   specific hardware/software platform is thus instantiated only once in a

given model instantiation, no matter how many data elements may be

declared with this type.

Public Attributes:

addrUnit : enumeration

15   Number of bits in storage addressable unit

   - bit/byte/word/dword

width : int

Number of addressable storage units in the described type. This can be 1,

20   8, 16, 32 bits.

alignment : int

Required alignment of type in address space - e.g. word aligned 32 bit

integer would have alignment of 4

nickname : int

25   Name of the base element. This should uniquely identify an instance of a

simple type to allow logic based on name rather than logic based on

combinations of attributes. E.g. "S390Binary31_0" for a 32 bit S/390

unscaled binary integer

bigEndian : boolean

30   True if this element is Big Endian format.

Stereotypes

bitModePad

Public Attributes:

16bit :

24bit :

31bit :

32bit :

64bit :

128bit :

signCoding

Note that this model does not include the following COBOL usages:

1   POINTER

PROCEDURE-POINTER

OBJECT REFERENCE

Public Attributes:

twosComplement :

onesComplement :

signMagnitude :

zoneSigns :

packedSigns :

unsignedBinary :

unsignedDecimal :

lengthEncoding

Public Attributes:

fixedLength :

lengthPrefixed :

nullTerminated :

45

floatType

      Public Attributes:

      Unspecified :

      IEEEextendedIntel :

5   For IEEE extended floats running on Intel Platform

      IEEEextendedAIX :

      For IEEE extended floats running on AIX Platform

      IEEEextendedOS390 :

      For IEEE extended floats running on OS/390 Platform

10   IEEEextendedAS400 :

      For IEEE extended floats running on AS400 Platform

      IEEEnonextended :

      For IEEE non-extended floats

      IBM390Hex :

15   For Hexadecimal floats running on IBM OS/390

      IBM400Hex :

      For Hexadecimal floats running on IBM AS400

accessor

20   Public Attributes:

      ReadOnly :

      WriteOnly :

      ReadWrite :

      NoAccess :

25

packedDecimalSign

      Public Attributes:

      MVS :

      MVSCustom :

30   NT/OS2/AIX :

46

5. Language Metamodels

5. b. PL/I Metamodel

The PLI language metamodel is used by enterprise application programs to define data structures (semantics) which represent connector interfaces. This metamodel is a MOF Class instance at the M2 level. FIGURE 13 illustrates a PL/I language metamodel, which is usable by application programs to define data structures which represent connector interfaces. This metamodel is an MOF class instance at the M2 level. FIGURE 14 illustrates the TDLangClassifier, the TDLanguageComposedType, and the TDLanguageElement for the PL/I Metamodel. FIGURE 15 illustrates enumerations of Mode Values, Base Values, LengthTypes, and StringTypeValues for the PL/I Metamodel.

5. b. i. PL/I Metamodel Specification

PLI

The PLI language metamodel is used by enterprise application programs to define data structures (semantics) which represent connector interfaces,

This language model for PL/I attempts to describe PL/I declares that have the storage class of either
PARAMETER, STATIC or BASED. CONTROLLED, AUTOMATIC and DEFINED are not supported.

In the PL/I languages, extents(that is string lengths, area sizes and array bounds) may, in general, be declared
as constants, as expressions to be evaluated at run-time, as asterisks, or as defined via the REFER option;
however, none of these choices are valid for all storage classes.

Based variables whose extents are not constant and not defined via the REFER option are excluded
from this model, as are parameters whose extents are specified via asterisks.

The INITIAL attribute (which is not valid for parameters in any case) will be ignored by the model.

47

PLISimpleType
> PLISimpleType is an abstract class that contains attributes shared by all simple types in the PLI   language.

5   Derived from PLIClassifier

PLIElement
> * A PLIDeclaration represents every data definition in the PLI program.
> * For example:

10   DCL 1 Employee,
>   2 Name,
>   3 Last char(29),
>   3 First char(29),
>   2 Deparment char(5);

15   where Eemployee, Name, Last, First, and Department are all PLI Elements.

        ntation.
        Derived from TDLangElement

20

        Public Attributes:
        level : String

25   **PLIComposedType**
> PLIComposedType is a collection of member elements that can be structure, unions, or elementary variables and arrays. PLIComposedType has a single aggregation to include all the elements that are a part of this composition.

30

> A structure is a data aggregate whose elements need not have identical attributes. Like an array, the entire tructure is given a name that can be used to refer to the entire aggregate of data. Unlike an array, however, each element of a structure also has a name.

35

> A structure has different levels. At the first level is the structure name called a major structure. At a deeper level are the names of substructures called minor structures. At the deepest are the element names called elementary names. An elementary name in a structure can represent an

40   array, in which case it is not an element variable, but an array variable.

> The organization of a structure is specified in a DECLARE statement through the use of level numbers preceding the associated names; level numbers must be integers. A major structure name is declared with the

48

level number 1. Minor structures and elementary names are declared with level numbers greater than 1.

The description of a major structure name is terminated by one of the following:

The declaration of another item with a level number 1
The declaration of another item with no level number
A semicolon terminating the DECLARE statement

A delimiter (usually a blank) must separate the level number and its associated name. For example, the items of a payroll can be declared as follows:

```
DECLARE 1 PAYROLL,
      2 NAME,
         3 LAST CHAR(20),
         3 FIRST CHAR(15),
      2 HOURS,
         3 REGULAR FIXED DEC(5,2),
         3 OVERTIME FIXED DEC(5,2),
      2 RATE,
         3 REGULAR FIXED DEC(3,2),
         3 OVERTIME FIXED DEC(3,2);
```

Indentation is only for readability. The statement could be written in a continuous string as DECLARE 1 PAYROLL, 2 NAME, 3 LAST CHAR(20), etc.

PAYROLL is declared as a major structure containing the minor structures NAME, HOURS, and RATE. Each minor structure contains two elementary names. You can refer to the entire structure by the name PAYROLL, or to portions of the structure by the minor structure names. You can refer to an element by referring to an elementary name.

The level numbers you choose for successively deeper levels need not be consecutive. A minor structure at level n contains all the names with level numbers greater than n that lie between that minor structure name and the next name with a level number less than or equal to n. PAYROLL might have been declared as follows:

```
DECLARE 1 PAYROLL,
      4 NAME,
```

49

5 LAST CHAR(20),
5 FIRST CHAR(15),
2 HOURS,
6 REGULAR FIXED DEC(5,2),
5 OVERTIME FIXED DEC(5,2),
2 RATE,
3 REGULAR FIXED DEC(3,2),
3 OVERTIME FIXED DEC(3,2);

This declaration results in exactly the same structuring as the previous declaration.
Derived from PLIClassifier, TDLangComposedType

Public Attributes:
Union : Boolean

PLIElementInitialValue
The INITIAL attribute specifies an initial value or values assigned to a variable at the time storage is allocated for it. Only
one initial value can be specified for an element variable. More than one can be specified for an array variable. A
structure or union variable can be initialized only by separate initialization of its elementary names, whether they are
element or array variables. The INITIAL attribute cannot be given to constants, defined data, noncontrolled parameters,
and non-LIMITED static entry variables.

The INITIAL attribute has three forms.

1.The first form, INITIAL, specifies an initial constant, expression, or function reference, for which the value is
   assigned to a variable when storage is allocated to it.

2.The second form, INITIAL CALL, specifies (with the CALL option) that a procedure is invoked to perform
   initialization. The variable is initialized by assignment during the execution of the called routine. (The routine is not
   invoked as a function that returns a value to the point of invocation.)

3.The third form, INITIAL TO, specifies that the pointer (or array of pointers) is initialized with the address of the

50

character string specified in the INITIAL LIST. The string also has the attributes indicated by the TO keyword.

5

Derived from TDLangElement

10   Public Attributes:
         initialValue : String
         ValueType : PLIInitalValueType

15   **PLIClassifier**
         PLI type
         Derived from TDLangClassifier

PLISourceText
20   This class contains the entire source code (including comments) and its
         associated line number.

         Public Attributes:
         source : String
25   **fileName : String**

PLIComputationalType
         Computational types represent values that are used in computionals to
30   produce a desired result. Arthmetic and string data consitute computional
         data

         Derived from PLISimpleType
35

PLIArithmeticType
         Aritmetic data are rational numbers

40   Derived from PLIComputationalType

Public Attributes:
mode : ModeValues

5   **PLIStringType**
a string is a sequence of contiguous characters, bit, cidechars, or graphics
that are treated a a single data item
Derived from PLIComputationalType

10   **PLIIntegerType**
Declared as Fixed Bin. with precision and scale factors. The prcision is
the total number of digits and the scale is the number of digits to the right
of the binary point.

15   Example

DCL bindata BINARY FIXED (7,3)

could be used to present the number 1011.111B that is 7 data bits, 3 of
20   which is to the right of the binary point.
Derived from PLIArithmeticType

Public Attributes:
precision : Integer
25   **scale : Integer**
Signed : Boolean
BigEndian : Boolean = true

30   **PLIFloatType**
has   precision = 2A decimal floating-point constant is a mantissa
followed by an exponent. The mantissa is a decimal fixed-point constant.
The exponent is the letter E followed by an optionally signed integer,
which specifies a power of ten. Decimal floating-point constants have a
35   precision (p), where p is the number of digits of the mantissa.
Examples are:

15E-23	has precision = 2
15E23	has precision = 2)
4E-3	has precision = 1
1.96E+07	has precision = 3
438E0	has precision = 3
3141593E-6	has precision = 7

52

.003141593E3   has precision = 9

The last two examples represent the same value (although with different precisions).

The data attributes for declaring decimal floating-point variables are DECIMAL and FLOAT. For example:

DECLARE LIGHT_YEARS DECIMAL FLOAT(5);

LIGHT_YEARS represents decimal floating-point data items with at least 5   decimal digits.

A binary floating-point constant is a mantissa followed by an exponent and the letter B. The mantissa is a binary fixed-point constant. The exponent is the letter E, followed by an optionally signed decimal integer, which specifies a power of two. Binary floating-point constants have a precision (p) where p is the number of binary digits of the mantissa. Examples are:

101101E5B has precision = 6

101.101E2B has precision = 6

11101E-28B has precision = 5

11.01E+42B has precision = 4

The data attributes for declaring binary floating-point variables are BINARY and FLOAT. For example:

DECLARE S BINARY FLOAT (16);

S represents binary floating-point data items with a precision of 16 binary digits.

The default precision is (21). The exponent cannot exceed three decimal digits. Binary floating-point data is stored as normalized hexadecimal

53

floating-point. If the declared precision is less than or equal to (21), short floating-point form is used. If the declared precision is greater than (21) and less than or equal to (53), long floating-point form is used. If the declared precision is greater than (53), extended floating-point form is used.

Derived from PLIArithmeticType

Public Attributes:
base : BaseValues
precision : Integer
IEEE : Boolean = false
BigEndian : Boolean = true

PLIPackedType

The data attributes for declaring decimal fixed-point variables are DECIMAL and FIXED. For example:

declare A fixed decimal (5,4);

specifies that A represents decimal fixed-point data of 5 digits, 4 of which are to the right of the decimal point.

These two examples:

declare B fixed (7,0) decimal;
declare B fixed decimal(7);

both specify that B represents integers of 7 digits.

This example

declare C fixed (7,-2) decimal;

specifies that C has a scaling factor of -2. This means that C holds 7 digits that range from -9999999*100 to 9999999*100, in increments of 100.

This example

declare D decimal fixed real(3,2);

54

specifies that D represents fixed-point data of 3 digits, 2 of which are fractional.

5

Derived from PLIArithmeticType

Public Attributes:

10 **precision : Integer**
scale : Integer

PLIPictureType

15 Numeric picture data is numeric data that is held in character form
Derived from PLIArithmeticType

Public Attributes:
pictureString : String

20

PLICodedStringType
Derived from PLIStringType

Public Attributes:

25 **type : StringTypeValues**
varying : LengthType

30 **PLIPictureStringType**
A character picture specification describes a fixed-length character data item, with the additional facility of indicating that any position in the data item can only contain characters from certain subsets of the complete set of available characters.

35

A character picture specification is recognized by the occurrence of an A or X picture specification character. The only valid characters in a character picture specification are X, A, and 9. Each of these specifies the presence of one character position in the character value, which can
40 contain the following:

55

X   Any character of the 256 possible bit combinations represented by the 8-bit byte.

A   Any alphabetic character, #, @, $, or blank.

9   Any digit, or blank. (Note that the 9 picture specification character in numeric character specifications is different in that the   corresponding character can only be a digit).

When a character value is assigned, or transferred, to a pictured character data item, the particular character in each position is
checked for validity, as specified by the corresponding picture specification character, and the CONVERSION condition is raised for an invalid character. (However, if you change the value either via record-oriented transmission or using an alias, there is no checking).
For example:

DECLARE PART# PICTURE 'AAA99X';
PUT EDIT (PART#) (P'AAA99X');

The following values are valid for PART#:

'ABC12M'
'bbb09/'
'XYZb13'

The following values are not valid for PART# (the invalid characters are underscored):

'AB123M'
'ABC1/2'
'Mb#A5;'

Derived from PLIStringType

Public Attributes:
pictureString : String

56

PLINonComputationalType
> Represents values that are used to control execution of your program. It consists of the following data types--area, entry, label, file, format, pointer, and offset.

5
> Derived from PLISimpleType

PLILabelType
> A label is a label constant or the value of a label variable.

10
> A label constant is a name written as the label prefix of a statement (other than PROCEDURE, ENTRY, PACKAGE, or FORMAT) so that during execution,
> program-control can be transferred to that statement through a reference to it. In the following line
15
> of code, for example, Abcde is a label constant.

> Abcde: Miles = Speed*Hours;

20
> The labelled statement can be executed either by normal sequential execution of instructions or by using the GO TO statement to transfer control to it from some other
> point in the program.

25
> A label variable can have another label variable or a label constant assigned to it. When such an assignment is made, the environment of the source label is assigned to
> the target. If you declare a static array of labels to have initial values, the array is treated as nonassignable.

30
> A label variable used in a GO TO statement must have as its value a label constant that is used in a block that is active at the time the GO TO is executed. Consider
> the following example:

35
> declare Lbl_x label;
> Lbl_a: statement;

> .

40
> .

> .

45
> Lbl_b: statement;

57

.

.

5

.

Lbl_x = Lbl_a;

.

10

.

.

15   go to Lbl_x;

Lbl_a and Lbl_b are label constants, and Lbl_x is a label variable. By
assigning Lbl_a to Lbl_x, the statement GO TO Lbl_x transfers control to
20 the Lbl_a
statement. Elsewhere, the program can contain a statement assigning
Lbl_b to Lbl_x. Then, any reference to Lbl_x would be the same as a
reference to Lbl_b.
This value of Lbl_x is retained until another value is assigned to it.

25

If a label variable has an invalid value, detection of such an error is not
guaranteed. In the following example, transfer is made to a particular
element of the array Z
based on the value of I.

30
    go to Z(I);

.

35   .

.

Z(1): if X = Y then return;

40

.

.

45   .

58

Z(2): A = A + B + C * D;

.

5   .

.

Z(3): A = A + 10;

10

If Z(2) is omitted, GO TO Z(I) when I=2 raises the ERROR condition. GO
TO Z(I) when I < LBOUND(Z) or I > HBOUND(Z) causes unpredictable
results if the
15   SUBSCRIPTRANGE condition is disabled.
Derived from PLINonComputationalType

PLIFormatType
The remote (or R) format item specifies that the format list in a FORMAT
20   statement is to be used.

The R format item and the specified FORMAT statement must be internal
to the same block, and they must be in the same invocation of that block.

25   A remote FORMAT statement cannot contain an R format item that
references itself as a label-reference, nor can it reference another remote
FORMAT statement that leads to the eferencing of the original FORMAT
statement.

30   Conditions enabled for the GET or PUT statement must also be enabled
for the remote FORMAT statement(s) that are referred to.

If the GET or PUT statement is the single statement of an ON-unit, that
statement is a block, and it cannot contain a remote format item.

35
For example:

DECLARE SWITCH LABEL;
GET FILE(IN) LIST(CODE);
40   IF CODE = 1
    THEN SWITCH = L1;
    ELSE SWITCH = L2;
GET FILE(IN) EDIT (W,X,Y,Z)
    (R(SWITCH));
45   L1: FORMAT (4 F(8,3));
L2: FORMAT (4 E(12,6));

59

SWITCH has been declared to be a label variable; the second GET statement can be made to operate with either of the two FORMAT statements.

Data format items describe the character or graphic representation of a single data item. They are:

A - Character
B - Bit
C - Complex
E - Floating point
F - Fixed point
G - Graphic
P - Picture

The remote format item specifies a label reference whose value is the label constant of a FORMAT statement located elsewhere. The FORMAT statement contains the remotely situated format items. The label reference item is: R.

The first data format item is associated with the first data list item, the second data format item with the second data list item, and so on. If a format list contains fewer data format items than there are items in the associated data list, the format list is reused. If there are excessive format items, they are ignored.

Suppose a format list contains five data format items and its associated data list specifies ten items to be transmitted. The sixth item in the data list is associated with the first data format item, and so forth. Suppose a format list contains ten data format items and its associated data list specifies only five items. The sixth through the tenth format items are ignored.

Example:

GET EDIT (NAME, DATA, SALARY)(A(N), X(2), A(6), F(6,2));

This example specifies that the first N characters in the stream are treated as a character string and assigned to NAME. The next 2 characters are skipped. The next 6 are assigned to DATA in character format. The next 6 characters are considered an optionally signed decimal fixed-point constant and assigned to SALARY.

60

Derived from PLINonComputationalType

PLIEntryType

Entry data can be an entry constant or the value of an entry variable.

An entry constant is a name written as a label prefix to a PROCEDURE or ENTRY statement, or a name declared with the ENTRY attribute and not the VARIABLE attribute, or the name of a mathematical built-in function.

An entry constant can be assigned to an entry variable. For example:

```
P: PROCEDURE;
DECLARE EV ENTRY VARIABLE,
   (E1,E2) ENTRY;


EV = E1;
CALL EV;
EV = E2;
CALL EV;
```

P, E1, and E2 are entry constants. EV is an entry variable. The first CALL statement invokes the entry point E1. The second CALL invokes the entry point E2.

The following example contains a subscripted entry reference:

```
DECLARE (A,B,C,D,E) ENTRY,
DECLARE F(5) ENTRY VARIABLE
     INITIAL (A,B,C,D,E);
DO I = 1 TO 5;
CALL F(I) (X,Y,Z);
END;
```

The five entries A, B, C, D, and E are each invoked with the parameters X, Y, and Z.

When an entry constant which is an entry point of an internal procedure is assigned to an entry variable, the assigned value remains valid only for as long as the block that the entry constant was internal to remains active (and, for recursive procedures, current).

ENTRYADDR built-in function and pseudovariable allows you to get or set the address of the entry point of a PROCEDURE or an

61

ENTRY in an entry variable. The address of the entry point is the address of the first instruction that would be executed if the entry were invoked. For example:

5

PTR1 = ENTRYADDR(ENTRY_VBL);

obtains the address of the entry point, and

10

ENTRYADDR(ENTRY_VBL) = PTR2;
Derived from PLINonComputationalType

Public Attributes:
Limited : Boolean

15

PLIAreaType

Area variables describe areas of storage that are reserved for the allocation of based variables. This reserved storage can be allocated to, and freed from, based variables by the ALLOCATE and FREE statements. Area variables can have any storage class and must be aligned.

20

When a based variable is allocated and an area is not specified, the storage is obtained from wherever it is available. Consequently, allocated based variables can be scattered widely throughout main storage. This is not significant for internal operations because items are readily accessed using the pointers. However, if these allocations are transmitted to a data set, the items have to be collected together. Items allocated within an area variable are already collected and can be transmitted or assigned as a unit while still retaining their separate identities.

25

30

You might want to identify the locations of based variables within an area variable relative to the start of the area variable. Offset variables are provided for this purpose.

35

An area can be assigned or transmitted complete with its contained allocations; thus, a set of based allocations can be treated as one unit for assignment and input/output while each allocation retains its individual identity.

40

The size of an area is adjustable in the same way as a string length or an array bound and therefore it can be specified by an expression or an asterisk (for a controlled area or parameter) or by a REFER option (for a based area).

45

62

The area size for areas that have the storage classes AUTOMATIC or CONTROLLED is given by an expression whose integer value specifies the number of

5

reserved bytes.

If an area has the BASED attribute, the area size must be an integer unless the area is a member of a based structure and the REFER option is used.

10

The size for areas of static storage class must be specified as an integer.

Examples of AREA declarations are:

DECLARE AREA1 AREA(2000),

15

AREA2 AREA;

In addition to the declared size, an extra 16 bytes of control information precedes the reserved size of an area. The 16 bytes contain such details as

20

the amount of storage in use.

The amount of reserved storage that is actually in use is known as the extent of the area. When an area variable is allocated, it is empty, that is, the area extent is zero. The maximum extent is represented by the area

25

size. Based variables can be allocated and freed within an area at any time during execution, thus varying the extent of an area.

When a based variable is freed, the storage it occupied is available for other allocations. A chain of available storage within an area is

30

maintained; the head of the chain is held within the 16 bytes of control information. Inevitably, as based variables with different storage requirements are allocated and freed, gaps occur in the area when allocations do not fit available spaces. These gaps are included in the extent of the area.

35

No operators, including comparison, can be applied to area variables.
Derived from PLINonComputationalType

PLIPointerType

40

A pointer variable is declared contextually if it appears in the declaration of a based variable, as a locator qualifier, in a BASED attribute, or in the SET option of an ALLOCATE, LOCATE, or READ statement. It can also be declared explicitly.
Derived from PLINonComputationalType

45

63

PLIFileType
>The FILE attribute specifies that the associated name is a file constant or file variable.

5

>The FILE attribute can be implied for a file constant by any of the file description attributes. A name can be contextually declared as a file constant through its appearance in the FILE option of any input or output statement, or in an ON statement for any input/output condition.
>Derived from PLINonComputationalType

10

PLIOffsetType
>You might want to identify the locations of based variables within an area variable relative to the start of the area variable. Offset variables are provided for this purpose.

15
>Derived from PLINonComputationalType

>Public Attributes:
>**BigEndian : Boolean = true**

20

PLIAlias
>An alias is a type name that can be used wherever an explicit data type is allowed. Using the DEFINE ALIAS statement, you can
>define an alias for a collection of data attributes. In this way, you can

25
>assign meaningful names to data types and improve the understandability of a program. By defining an alias, you can also provide a shorter notation for a set of data attributes, which can
>decrease typographical errors.

30
>Example

```
define alias Name   char(31) varying;
define alias Salary fixed dec(7);   /* real by default     */
define alias Zip    char(5)         /* nonvarying by default */
```

35

>Derived from PLINamedType
Attribute
>The attributes that can be specified are any of the attributes for variables

40
>that can be returned by a function (for example, those attributes valid in the RETURNS option and attribute). That is, you cannot specify an alias for an array or a structured attribute list. However, you can specify an alias for a type that is defined in a

64

DEFINE ORDINAL, or DEFINE STRUCTURE statement or in another DEFINE ALIAS statement. Also, as in the RETURNS option and attribute,
any string lengths or area sizes must be restricted expressions.

5

Missing data attributes are supplied using PL/I defaults.
Public Attributes:
attribute : String

10

PLIOrdinal

An ordinal is a named set of ordered values. Using the DEFINE ORDINAL statement, you can define an ordinal and assign meaningful names to be used to refer to that set. For example, you can
15   define an ordinal called "color". The "color" ordinal could include the members "red", "yellow", "blue", etc. The members of the "color" set can then be referred to by these names instead of by their associated fixed binary value, making code much more self-documenting. Furthermore, a variable declared with the ordinal type
20   can be assigned and compared only with an ordinal of the same type, or with a member of that ordinal type. This automatic checking provides for better program reliability.

In the following example, Red has the value 0, Orange has the value 1, etc.
25   But Low has the value 2 and Medium has the value 3.

Example

30

```
define ordinal Color ( Red,        /* is 0, since VALUE is omitted */
                       Orange,
                       Yellow,
                       Green,
35                     Blue,
                       Indigo,
                       Violet );

       define ordinal Intensity ( Low    value(2),
40                                 Medium,
                                   High   value(5));
```
Derived from PLINamedType

Public Attributes:
precision : int
isSigned : boolean

5

OrdinalValue

OrdinalValue specifies the value of a particular member within the set. If the VALUE attribute is omitted for the first member,
a value of zero is used. If the VALUE attribute is omitted for any other
10   member, the next greater integer value is used.

The value in the given (or assumed) VALUE attribute must be an integer, can be signed, and must be strictly increasing. The value in
the given (or assumed) VALUE attributed may also be specified as an XN
15   constant.

Example
define ordinal Intesity (low vcalue(2), middle, high value (5));

20   will giive low a value of 2, middle a value of 3, and high a value of 5.
Public Attributes:
name : String
value : int

25

PLIStructure

A structure is a collection of member elements that can be structure,
unions, or elementary variables and arrays.

30

Unions are a collection of member elements that overlay each other,
occupying the same storage. Unions
can be used to declare variant records that typically contain a common
part, a selector part, and a variant part

35

DEFINE STRUCTURE statement is used to specify a named structure or
union. Structures are stronly typed. That is only variables declared as
structures can be assigned to variables or parameters declared as
structures.

40

Derived from PLINamedType, PLIComposedType

66

PLIHandleType
> use HANDLE to declare a variable as a pointer to a structure type.
> Handle are strongly typed - they can be assigned to or compared with
> handles for the same structure type

5
> Derived from PLINonComputationalType

> Public Attributes:
> **structure : tPLIStructure**

10

PLINamedType
> PL/I allows you to define your own named types using the built-in data
> types. These include user-defined types (aliases,ordinals, structures, and
> unions), declarations of variables with these types, handles, and type
15
> functions.
> Derived from PLIClassifier

PLIArray
> An array is an n-dimensional collection of elements that have idenical
20
> attributes.

> Examples

> dcl List fixed decimal(3) dimension(8)

25

> is a 1 dimensional array with 8 fixed-point decimal elements

> dcl table (4,2) fixed dec (3)

30
> is a 2 dimensional array of three fixed -point decimal elements the 2
> dimensions of table have lower bounds of 1 and upper bounds of 4 for 1
> dimension
> and lower bounds of 1 and upper bounds of 2 for the other

35
> The REFER option is used in the declaration of a based structure to
> specify that, on allocation of the structure, the value of an expression is
> assigned to a variable in the structure and represents the length, bound, or
> size of another variable in the structure. The syntax for a length, bound, or
> size with a REFER option is:

40
> expression REFER (variable)

> The variable, known as the object of the REFER option, must be a
> member of the declared structure. The REFER object must conform to the
45
> following rules:

It must be REAL FIXED BINARY(p,0).

It must precede the first level-2 element that has the REFER option or contains an element that has the REFER option.

It must not be locator-qualified or subscripted.

It must not be part of an array.

For example:

```
DECLARE 1 STR BASED(P),
       2 X FIXED BINARY,
       2 Y (L REFER (X)),
       L FIXED BINARY INIT(1000);
```

This declaration specifies that the based structure STR consists of an array Y and an element X. When STR is allocated, the upper bound is set to the current value of L which is assigned to X. For any other reference to Y, such as a READ statement that sets P, the bound value is taken from X.

If both the REFER option and the INITIAL attribute are used for the same member, initialization occurs after the object of the REFER has been assigned its value.

Any number of REFER options can be used in the declaration of a structure, provided that at least one of the following restrictions is satisfied:

All objects of REFER options are declared at logical level two, that is, not declared within a minor structure. For example:

```
DECLARE 1 STR BASED,
       2 (M,N),
       2 ARR(I REFER (M),
        J REFER(N)),
       2 X;
```

When this structure is allocated, the values assigned to I and J set the bounds of the two-dimensional array ARR.

The structure is declared so that no padding between members of the structure can occur. For example:

68

```
            DECLARE 1 STR UNAL BASED (P),
                2 B FIXED BINARY,
                2 C,
                    3 D FLOAT DECIMAL,
                    3 E (I REFER (D))
                        CHAR(J REFER (B)),
                2 G FIXED DECIMAL;
```

All items require only byte alignment because this structure has the
UNALIGNED attribute. Therefore, regardless of the values of B and D
(the REFER objects) no padding occurs. Note that D is declared within a
minor structure.

If the REFER option is used only once in a structure declaration, the
two preceding restrictions can be ignored provided that:

For a string length or area size, the option is applied to the last
element of the structure.

For an array bound, the option is applied either to the last element of
the structure or to a minor structure that contains the last element. The
array bound must be the upper bound of the leading dimension. For
example:

```
            DCL 1 STR BASED (P),
                2 X FIXED BINARY,
                2 Y,
                    3 Z FLOAT DECIMAL,
                    3 M FIXED DECIMAL,
                2 D (L REFER (M)),
                    3 E (50),
                    3 F (20);
```

The leading dimension of an array can be inherited from a higher
level. If we had declared STR(4) in the above example, the leading
dimension would be inherited from STR(4) and so it would not be possible
to use the REFER option in D.

This declaration does not satisfy the two previous bulleted
restrictions. The REFER object M is declared within a minor structure and
padding occurs. However, this restriction is satisfied because the REFER
option is applied to a minor structure that contains the last element.

The following example is also valid:

69

```
DCL 1 STR BASED(P),
    2 X FIXED BINARY(31),
    2 Y,
      3 Z FLOAT DECIMAL,
      3 M FIXED BINARY(31),
      3 N FIXED BINARY(31),
    2 D,
      3 E(50),
      3 F(L REFER (M))
        CHAR(K REFER (N));
```

If the value of the object of a REFER option varies during the program then:

The structure must not be freed until the object is restored to the value it had when allocated.

The structure must not be written out while the object has a value greater than the value it was allocated.

The structure can be written out when the object has a value equal to or less than the value it had when allocated. The number of elements, the string length, or area size actually written is that indicated by the current value of the object. For example:

```
DCL 1 REC BASED (P),
    2 N,
    2 A (M REFER(N)),
    M INITIAL (100);
ALLOCATE REC;
N = 86;
WRITE FILE (X) FROM (REC);
```

86 elements of REC are written. It would be an error to attempt to free REC at this point, since N must be restored to the value it had when allocated (that is, 100). If N is assigned a value greater than 100, an error occurs when the WRITE statement is encountered.

When the value of a refer object is changed, the next reference to the structure causes remapping. For example:

```
DCL 1 A BASED(P),
```

70

```
        2 B,
        2 C (I REFER(B)),
        2 D,
    I INIT(10);

        ALLOCATE A;
        B = 5;
```

The next reference to A after the assignment to B remaps the structure to reduce the upper bound of C from 10 to 5, and to allocate to D storage immediately following the new last element of C. Although the structure is remapped, no data is reassigned--the contents of the part of storage originally occupied by the structure A are unchanged. If you do not consider remapping, errors can occur. Consider the following example, in which there are two REFER options in the one structure:

```
    DCL 1 A BASED (P),
        2 B FIXED BINARY (15,0),
        2 C CHAR (I1 REFER (B)),
        2 D FIXED BINARY (15,0),
        2 E CHAR (I2 REFER (D)),
    (I1,I2) INIT (10);

    ALLOCATE A;
    B = 5;
```

The mapping of A with B=10 and B=5, respectively, is as follows:

```
    |B|    C    |D|    E    |
    |__|_____|__|_____|

    |B| C |D|   E
    |__|____|__|__|_____
```

D now refers to data that was originally part of that assigned to the character-string variable C. This data is interpreted according to the attributes of D--that is, as a fixed-point binary number--and the value obtained is the length of E. Hence, the length of E is unpredictable.

PLIFixedLengthString

Derived from PLICodedStringType

71

Public Attributes:
length : Integer

PLIVariableLengthString
Derived from PLICodedStringType

Public Attributes:
lengthToAllocate : String

PLIFixedLengthArea
Derived from PLIAreaType
Private Attributes:
length : Integer

PLIVariableLengthArea
The size of an area is adjustable in the same way as a string length or an array bound and therefore it can be specified by an expression or an asterisk (for a controlled area or parameter) or by a REFER option (for a based area).
Derived from PLIAreaType

Private Attributes:
lengthToAllocate : String

PLIFixedBoundArray
Derived from PLIArray

Public Attributes:
lbound : Integer
ubound : Integer

PLIFixedLboundArray
Derived from PLIArray

72

Public Attributes:
lbound : Integer
HboundtoAllocate : String

5

PLIHboundArray
Derived from PLIArray

Public Attributes:
10 **LboundtoAllocate : String**
ubound : Integer

PLIVariableBoundArray
15 Derived from PLIArray

Public Attributes:
LboundToAllocate : String
HboundToAllocate : String

20 **TOTALS:**

1 Logical Packages
39 Classes

LOGICAL PACKAGE STRUCTURE

Logical View
25   PLI
            TDLang

6.   Illustrative Applications of the Common Application Metamodel and System

30

Various complex transaction, occurring across a plurality of individual platforms require

the seamlessness and transparency of the Common Application Metamodel. These

transactions include "rich" transactions, high level group ware, and financial transactions.

35 FIGURE 16 illustrates a simplified network configuration for a "rich" transaction where,

for example, an order is entered at a terminal, and is passed to and through a Web server

73

to a manufacturer's application server. The manufacturer's application server searches through it's own database server, as well as its vendors' dissimilar and incompatible database servers and databases, transparently connected by the connectors described herein, querying for statuses, prices, and delivery dates, of components, and placing

5   orders for needed components to satisfy the order.

Strictly for purposes of illustration and not limitation, a rich transaction is illustrated with the purchase of an automobile, from configuring and ordering the new automobile through financing the automobile, collecting the elements and components the new

10   automobile, assembling the automobile, and delivering it to the customer. In configuring the automobile, consider the inclusion of, e.g., a traction control module, which may require the inclusion of one sub-set of engines, while precluding the inclusion of another sub-set of engines, for example, for technical or performance reasons. Similarly, the selection of one color may preclude the selection of certain upholstery combinations, for

15   example, for reasons of inventory or availability. Of course, if you don't select an engine and a transmission, you don't have an automobile. Certain choices must be made. The problem is one of "If you pick `X`, you must also pick `Y` and `Z`, but if you pick `Y`, you can not get `A` or `B`, and you have already selected `A`." That is, selection of one component or sub-system may remove some previously selected components or sub-

20   systems from the system. After configuring the car, frequently based on the availability of selected elements in the supply pipeline, the order is sent to the manufacturer's server for fulfillment, including manufacturing scheduling. The manufacturer would query the vendors in the supply chain, for example, a transmission vendor, who would, in turn, query controller vendors, gear vendors, housing vendors, hydraulic conduit vendors, and

25   the like. These vendors would, in turn query their sub-vendors.

Using the connector method and system, the dealer and customer would configure and order the car at a terminal 3901. The configuration and order entry would be sent to the manufacturer's server 3911. The manufacturer's server queries the servers 3921, 3931,

30   and 3441, (where server 3921 could be the server for the manufacturer's in house components business and for its purchasing function) or its direct vendors, who in turn

74

query third sub-vendors, 3923, 3933, and 3933. The queries could be in the form of "fill in the blank" SQL queries, subject to access authorizations. The sub-vendor servers, 3923, 3933, and 3443 specialized views to the vendor's servers 3921, 3931, 3441, as clients, which in turn would present specialized views to the manufacturer's server 3911

5 as a client. The data returned to the manufacturer's server 3911 could be optimized with optimization and production scheduling applications, to present a serial number and delivery date to the buyer at the dealer's server.

A further application of the connectors of the present invention is in connection with

10 groupware, and especially linking "islands of groupware" together. "Groupware" is a broad term applied to technology that is designed to facilitate the work of groups. Groupware technology may be used to communicate, cooperate, coordinate, solve problems, compete, or negotiate. A Groupware suite of applications is comprised of programs that help people work together collectively, even if located remotely from each

15 other. Groupware services can include the sharing of calendars, collective writing, e-mail handling, shared database access, electronic meetings, video conferencing, and other activities.

FIGURE 17 illustrates a group ware session spread across multiple group ware

20 applications running on multiple, disparate platforms, and connected by the connectors described herein. Specifically illustrated are two groupware systems 4010 and 4020. Each system contains e-mail enables applications, 4013 and 4023, such as e-mail, schedule/calendar, word processor, spread sheet, graphics, and CTI (computer telephony integration). These are supported by message API's 4015, 4025 and operating systems,

25 4017 and 4027, and local groupware servers 4011 and 4021. Each of the local groupware servers, 4011 and 4021, has a connector of the present invention, an e-mail server, an e-mail data base, a directory server, and a replication database. The two groupware servers 4011 and 4021 communicate over a telecommunications medium, as the Net, a LAN, a WAN, or even the PSTN.

30

E-mail is by far the most common and basic groupware application. While the basic technology is designed to pass simple messages between people, even relatively simple e-mail systems today typically include features for forwarding messages, filing messages, creating mailing groups, and attaching files with a message. Other features that have been

5   explored include content based sorting and processing of messages, content based routing of messages, and structured communication (messages requiring certain information).

Workflow systems are another element of groupware. Work flow systems allow documents to be routed through organizations through a relatively-fixed process. A

10   simple example of a workflow application is an expense report in an organization: an employee enters an expense report and submits it, a copy is archived then routed to the employee's manager for approval, the manager receives the document, electronically approves it and sends it on and the expense is registered to the group's account and forwarded to the accounting department for payment. Workflow systems may provide

15   features such as routing, development of forms, and support for differing roles and privileges.

Hypertext is a system for linking text documents to each other, with the Web being an obvious example. However, whenever multiple people author and link documents, the

20   system becomes group work, constantly evolving and responding to others' work. Another common multi-user feature in hypertext (that is not found on the Web) is allowing any user to create links from any page, so that others can be informed when there are relevant links that the original author was unaware of.

25   Group calendaring is another aspect of groupware and facilitates scheduling, project management, and coordination among many people, and may provide support for scheduling equipment as well. Typical features detect when schedules conflict or find meeting times that will work for everyone. Group calendars also help to locate people.

30   Collaborative writing systems may provide both realtime support and non-realtime support. Word processors may provide asynchronous support by showing authorship and

76

by allowing users to track changes and make annotations to documents. Authors collaborating on a document may also be given tools to help plan and coordinate the authoring process, such as methods for locking parts of the document or linking separately-authored documents. Synchronous support allows authors to see each

5   other's changes as they make them, and usually needs to provide an additional communication channel to the authors as they work.

Group may be synchronous or real time, such as shared whiteboards that allow two or more people to view and draw on a shared drawing surface even from different locations.

10   This can be used, for instance, during a phone call, where each person can jot down notes (e.g. a name, phone number, or map) or to work collaboratively on a visual problem. Most shared whiteboards are designed for informal conversation, but they may also serve structured communications or more sophisticated drawing tasks, such as collaborative graphic design, publishing, or engineering applications. Shared whiteboards can indicate

15   where each person is drawing or pointing by showing telepointers, which are color-coded or labeled to identify each person.

A further aspect of groupware is video conferencing. Video conferencing systems allow two-way or multi-way calling with live video, essentially a telephone system with an

20   additional visual component, with time stamping for coordination.

Decision support systems are another aspect of groupware and are designed to facilitate groups in decision-making. They provide tools for brainstorming, critiquing ideas, putting weights and probabilities on events and alternatives, and voting. Such systems

25   enable presumably more rational and evenhanded decisions. Primarily designed to facilitate meetings, they encourage equal participation by, for instance, providing anonymity or enforcing turn-taking.

Multi-player games have always been reasonably common in arcades, but are becoming

30   quite common on the Internet. Many of the earliest electronic arcade games were multi-user, for example, Pong, Space Wars, and car racing games. Games are the prototypical

77

example of multi-user situations "non-cooperative", though even competitive games require players to cooperate in following the rules of the game. Games can be enhanced by other communication media, such as chat or video systems.

5 Some product examples of groupware include Lotus Notes and Microsoft Exchange, both of which facilitate calendar sharing, e-mail handling, and the replication of files across a distributed system so that all users can view the same information.

What makes the connectors of the present invention particularly attractive for groupware
10 applications is the diversity of groupware offerings and group server architectures, implementations, and platforms. The connector of the invention can act as a front end or gateway to the groupware server.

FIGURE 18 illustrates a commercial transaction where real goods are shipped from a
15 seller to a buyer, and various forms of electronic payment and secured electronic payment are used by the buyer to pay the seller, with banks and financial institutions connected through the connectors described herein. Specifically, a merchant or manufacturer 4101 sells a product to a customer 4103 that he has no "history" with. The product is shipped 4605. However, the buyer 4103 does not wish to be parted from his money until the
20 goods are received, inspected, approved, and "accepted", while the seller 4101 does not want to give up control of the goods until he has been paid. This fundamental commercial conflict has led to various paradigms, most involving hard copy "near moneys" or instruments of one form or another. Today, the financial transactions are most frequently those involving electronic fund transfers, and electronic versions of notes, instruments,
25 negotiable instruments, documentary drafts, payment orders, letters of credit, warehouse receipts, delivery orders, bills of lading, including claims on goods, that is, documents of title that purport to transfer title or physical custody of goods to the bearer or to a named person, and security interests in goods. Typically, the customer 4103 executes an instrument in favor of the seller 4101, directing the buyer's bank 4121 to pay buyer's
30 money to the seller 4101 through seller's bank 4125. Normally, this is a simple electronic transaction between buyers and sellers who have dealt with each other before, dealing

78

through banks or financial intermediaries who have dealt with each other before, and who are using the same or compatible software applications. However, in the extraordinary case where these preconditions are not satisfied, the connectors of the invention facilitate the electronic, bank-to-bank, side of the transaction.

5

While the invention has been described and illustrated with respect to applications having a single level of application program interfaces or converters, it is, of course, to be understood that such application program interfaces or converters may be present at various levels in the processing hierarchy, for example between Web Clients and Web

10 servers, between web servers and application servers, between application servers and database servers, and between application servers or database servers or both and various specialized repositories.

It is also to be understood, that while the invention has been described and illustrated

15 with respect to certain preferred embodiments and exemplification having individual clients and individual servers, there may be multiple clients, multiple servers, and applications that function as both clients and servers, as exemplified by groupware applications, and there might be multiple parallel lines and/or multiple hierarchical levels of application servers, data servers, and databases, as in systems for rich transactions.

20

While the invention has been described with respect to certain preferred embodiments and exemplifications, it is not intended to limit the scope of the invention thereby, but solely by the claims appended hereto.